Mixture Modeling

Michael Betancourt

October 2024

Table of contents

Sometimes a single probabilistic model just isn't enough. When a behavior of interest might be consistent with multiple probabilistic models we need to incorporate each of those possibilities into the final model. If we don't know which of those possible probabilistic is responsible for a particular instance of that behavior then the only way to built a consistent joint model is with *mixture modeling*.

Mixture modeling allows us to, for example, account for undesired contamination in observations, such as an irreducible background that overlaps with a desired signal. By mixing together probabilistic models for the signal and background events we can learn not only their individual behaviors but also their relative prevalence across the observed data.

In this chapter we'll review the mathematical foundations and general implementation of mixture models, including the potential for frustrating inferential behaviors. I will illustrate these concepts with a series of demonstrative analyses.

1 Implementing Mixture Models

From a mathematical perspective mixing multiple probabilistic models together is relatively straightforward. We have to do a bit of work, however, to derive an implementation that performs well in practice.

1.1 Categorical Implementations

Consider an mathematical space X and a collection of K component probability distributions specified with the probability density functions, $p_k(x)$, each providing a different model for a behavior of interest. To simplify the initial presentation we will assume that each component probability distribution is fixed so that there no model configuration variables to consider.

The most straightforward way to model a behavior that can arise from any of these component models is to introduce a categorical variable that labels the responsible component,

$$z \in \{1, \ldots, k, \ldots, K\},$$

and a corresponding probabilistic model,

$$p(z = k \mid \lambda_1, \ldots, \lambda_K) = \lambda_k,$$

where the categorical probabilities λ_k from a simplex configuration,

$$\begin{aligned} 0 \leq \lambda_k \leq 1 \\ \sum_{k=1}^{K} \lambda_k = 1. \end{aligned}$$

Each instance of this behavior is then modeled with a value $x \in X$, an assignment z, and the joint model

$$\begin{aligned} p(x, z \mid \lambda_1, \ldots, \lambda_K) &= p(x \mid z)\, p(z \mid \lambda_1, \ldots, \lambda_K) \\ &= p_z(x)\, \lambda_z. \end{aligned}$$

If we know the responsible component model then the component assignment variable z will be known. If we do not know z, however, then all we can do is infer it from any particular value x. Assuming that the component probabilities and component models are known the consistent component assignment are given by an application of Bayes' Theorem,

$$\begin{aligned} p(z \mid \tilde{x}, \lambda_1, \ldots, \lambda_K) &= \frac{p(\tilde{x}, z \mid \lambda_1, \ldots, \lambda_K)}{\sum_{k=1}^{K} p(\tilde{x}, k \mid \lambda_1, \ldots, \lambda_K)} \\ &= \frac{p_z(\tilde{x})\, \lambda_z}{\sum_{k=1}^{K} p_k(\tilde{x})\, \lambda_k}. \end{aligned}$$

When modeling multiple, independent instances of a behavior we have to consider possible heterogeneity in the categorical variables.

If all of the categorical variables are modeled with the same component probabilities then the joint model becomes

$$\begin{aligned} p(x_1, z_1, \ldots, x_N, z_N \mid \lambda_1, \ldots, \lambda_K) &= \prod_{n=1}^{N} p(x_n \mid z_n)\, p(z_n \mid \lambda_1, \ldots, \lambda_K) \\ &= \prod_{n=1}^{N} p_{z_n}(x_n)\, \lambda_{z_n}. \end{aligned}$$

On the other hand if the component probabilities can vary across observations then we need a separate simplex configuration for each observation,

$$\lambda_n = (\lambda_{n,1}, \ldots, \lambda_{n,K}),$$

and the joint model becomes

$$\begin{aligned} p(x_1, z_1, \ldots, x_N, z_N \mid \lambda_1, \ldots, \lambda_N) &= \prod_{n=1}^{N} p(x_n \mid z_n) \, p(z_n \mid \lambda_n) \\ &= \prod_{n=1}^{N} p_{z_n}(x_n) \, \lambda_{n, z_n}. \end{aligned}$$

In either case inferences of the individual z_n depend on only the value of the corresponding x_n,

$$p(z_n \mid \tilde{x}_n, \lambda_1, \ldots, \lambda_K) = \frac{p_{z_n}(\tilde{x}_n) \, \lambda_{z_n}}{\sum_{k=1}^{K} p_k(\tilde{x}_n) \, \lambda_k}$$

or

$$p(z_n \mid \tilde{x}_n, \lambda_{n,1}, \ldots, \lambda_{n,K}) = \frac{p_{z_n}(\tilde{x}_n) \, \lambda_{n, z_n}}{\sum_{k=1}^{K} p_k(\tilde{x}_n) \, \lambda_{n,k}},$$

again assuming fixed component probabilities and component probability distributions.

1.2 Marginal Implementations

In practice unknown categorical variables can quickly become a nuisance. Not only do they introduce a new variable that we have to infer for each observation but also those variables are discrete which limits the available computational tools. Specifically we cannot use gradient-based methods like Hamiltonian Monte Carlo to explore posterior distributions over the component assignments and component probabilities, let alone any configuration parameters for the component models, at the same time.

Fortunately marginalizing unknown categorical assignments out of the joint model is straightforward.

1.2.1 Single Observation

For a single instance marginalizing an unknown component assignment requires summing over the K possible values,

$$\begin{aligned} p(x \mid \lambda_1, \ldots, \lambda_K) &= \sum_{k=1}^{K} p(x, z = k \mid \lambda_1, \ldots, \lambda_K) \\ &= \sum_{k=1}^{K} p(x \mid z = k) \, p(z = k \mid \lambda_1, \ldots, \lambda_K) \\ &= \sum_{k=1}^{K} p_k(x) \, \lambda_k, \end{aligned}$$

which is often written as

$$p(x \mid \lambda_1, \ldots, \lambda_K) = \sum_{k=1}^{K} \lambda_k \, p_k(x).$$

In other words all we have to do to eliminate a categorical variable is add together all of the the component probability density functions weighted by the component probabilities.

1.2.2 Multiple Homogeneous Observations

Given multiple, independent instances with the same component probabilities we can marginalize each component individually,

$$\begin{aligned} p(x_1, \ldots, x_N \mid \lambda_1, \ldots, \lambda_K) &= \prod_{n=1}^{N} p(x_n \mid \lambda_1, \ldots, \lambda_K) \\ &= \prod_{n=1}^{N} \sum_{k=1}^{K} p_k(x_n) \, \lambda_k \\ &= \prod_{n=1}^{N} \sum_{k=1}^{K} \lambda_k \, p_k(x_n). \end{aligned}$$

1.2.3 Multiple Heterogeneous Observations

Accounting for heterogeneity in the component probabilities is a bit more cumbersome,

$$\begin{aligned} p(x_1, \ldots, x_N \mid \lambda_1, \ldots, \lambda_N) &= \prod_{n=1}^{N} p(x_n \mid \lambda_n) \\ &= \prod_{n=1}^{N} \sum_{k=1}^{K} p_k(x_n) \, \lambda_{n,k}. \end{aligned}$$

Interestingly we can sometimes simplify the heterogeneous model even further by marginalizing out not only the individual categorical variables but also the individual categorical probabilities!

Specifically if the individual categorical probabilities are independent and identically distributed then any consistent probabilistic model for their behavior will be of the form

$$p(\lambda_n, \ldots, \lambda_n) = \prod_{n=1}^{N} p(\lambda_n).$$

In this case the expectation value of any component probability is the same for all observations,

$$\begin{aligned}
\int \mathrm{d}\lambda_1 \cdots \mathrm{d}\lambda_N p(\lambda_1, \ldots, \lambda_N)\, \lambda_{n,k} &= \int \mathrm{d}\lambda_1 \cdots \mathrm{d}\lambda_N \left[\prod_{n=1}^{N} p(\lambda_n) \right] \lambda_{n,k} \\
&= \prod_{n' \neq 1}^{N} \int \mathrm{d}\lambda_{n'} p(\lambda_n) \cdot \int \mathrm{d}\lambda_n \, p(\lambda_n)\, \lambda_{n,k} \\
&= \prod_{n' \neq 1}^{N} 1 \cdot \int \mathrm{d}\lambda_n \, p(\lambda_n)\, \lambda_{n,k} \\
&= \int \mathrm{d}\lambda_n \, p(\lambda_n)\, \lambda_{n,k} \\
&\equiv \omega_k.
\end{aligned}$$

Moreover these individual expectation values form their own simplex configuration: the bounds on each $\lambda_{n,k}$ imply that

$$0 \leq \omega_k \leq 1$$

while

$$\begin{aligned}
\sum_{k=1}^{K} \omega_k &= \sum_{k=1}^{K} \int \mathrm{d}\lambda_n \, p(\lambda_n)\, \lambda_{n,k} \\
&= \int \mathrm{d}\lambda_n \, p(\lambda_n) \sum_{k=1}^{K} \lambda_{n,k} \\
&= \int \mathrm{d}\lambda_n \, p(\lambda_n)\, 1 \\
&= 1.
\end{aligned}$$

Under these assumptions the marginal mixture model is given by

$$\begin{aligned}
p(x_1, \ldots, x_N) &= \int \mathrm{d}\lambda_1 \cdots \mathrm{d}\lambda_N p(x_1, \ldots, x_N, \lambda_1, \ldots, \lambda_N) \\
&= \int \mathrm{d}\lambda_1 \cdots \mathrm{d}\lambda_N \left[\prod_{n=1}^{N} \sum_{k=1}^{K} p_k(x_n)\, \lambda_{n,k} \right] \prod_{n=1}^{N} p(\lambda_n) \\
&= \prod_{n=1}^{N} \int \mathrm{d}\lambda_n \left[\sum_{k=1}^{K} p_k(x_n)\, \lambda_{n,k} \right] p(\lambda_n) \\
&= \prod_{n=1}^{N} \sum_{k=1}^{K} \left[p_k(x_n) \int \mathrm{d}\lambda_n \, p(\lambda_n)\, \lambda_{n,k} \right] \\
&= \prod_{n=1}^{N} \sum_{k=1}^{K} \left[p_k(x_n)\, \omega_k \right].
\end{aligned}$$

In other words the heterogeneous model reduces to the same form as the homogeneous mixture model, only with the categorical probabilities

$$\omega = (\omega_1, \ldots, \omega_K)!$$

Critically the new categorical probabilities ω_k can no longer be interpreted as the probability of any *individual* instance of the target behavior arising from a particular component model. Instead they capture the *proportion* of the ensemble of instances that arises from each component model.

1.3 Numerically Stable Marginal Implementations

Most probabilistic programming languages, for example the Stan modeling language, work not with probability density functions $p(x)$ but rather log probability density functions $\log \circ p(x)$. Consequently in order to implement a mixture model in these languages we need to evaluate

$$\begin{aligned}
\log \circ p(x_1, \ldots, x_N \mid \lambda_1, \ldots, \lambda_K) &= \log \left(\prod_{n=1}^{N} \sum_{k=1}^{K} \lambda_k \, p_k(x_n) \right) \\
&= \sum_{n=1}^{N} \log \left(\sum_{k=1}^{K} \lambda_k \, p_k(x_n) \right) \\
&= \sum_{n=1}^{N} \log \left(\sum_{k=1}^{K} \lambda_k \, \exp(\log \circ p_k(x_n)) \right) \\
&= \sum_{n=1}^{N} \log \left(\sum_{k=1}^{K} \exp(\log(\lambda_k) + \log \circ p_k(y_n)). \right)
\end{aligned}$$

In other words for each observation we need to exponentiate a term for each component, evaluate the sum of the exponentials, and then apply the natural logarithm function.

This composite operation is often referred to as the "log sum exp" function,

$$\text{log-sum-exp}(v_1, \ldots, v_K) = \log \left(\sum_{k=1}^{K} \exp(v_k) \right).$$

Implementing this operation on computers is complicated by the limitations of floating point arithmetic. In particular the intermediate terms $\exp(v_k)$ are prone to overflowing to floating point infinity and corrupting the calculation before the final logarithm can calm things down.

Fortunately the properties of logarithms and exponentials allow us to implement the log sum exp operation in a variety of ways. Specifically we can always factor out any one component

without affecting the final value,

$$\begin{aligned}
\text{log-sum-exp}(v_1, \ldots, v_K) &= \log \left(\sum_{k=1}^{K} \exp(v_k) \right) \\
&= \log \left(\exp(v_{k'}) + \sum_{k \neq k'} \exp(v_k) \right) \\
&= \log \left(\exp(v_{k'}) \right) \left(1 + \sum_{k \neq k'} \frac{\exp(v_k)}{\exp(v_{k'})} \right) \\
&= \log \left(\exp(v_{k'}) \right) \left(1 + \sum_{k \neq k'} \exp(v_k - v_{k'}) \right) \\
&= \log \exp(v_{k'}) + \log \left(1 + \sum_{k \neq k'} \exp(v_k - v_{k'}) \right) \\
&= v_{k'} + \log \left(1 + \sum_{k \neq k'} \exp(v_k - v_{k'}) \right).
\end{aligned}$$

If we factor out the largest component value then

$$v_{k'} \geq v_{k \neq k'},$$

the input to each exponential will always be less than or equal to one, and the calculation will never encounter floating point overflow. Conveniently most computational libraries provide log sum exp function implementations that automatically factor out the largest value and ensure stable numerical computation. Because of this we need to worry about numerical stability only when implementing operations like these ourselves.

For example in Stan we can avoid any numerical issues by using the built-in `log_sum_exp` function.

```
// Loop over observations
for (n in 1:N) {
  target += log_sum_exp(log(lambda[1]) + foo1_lpdf(x[n]),
                        log(lambda[2]) + foo2_lpdf(x[n]),
                        log(lambda[3]) + foo3_lpdf(x[n]),
                        log(lambda[4]) + foo4_lpdf(x[n]));
}
```

Conveniently this function is also vectorized so we can call it with a single vector argument.

```
// Loop over observations
for (n in 1:N) {
  vector[4] lpds = [ foo1_lpdf(x[n]), foo2_lpdf(x[n]),
                     foo3_lpdf(x[n]), foo4_lpdf(x[n]) ]';
  target += log_sum_exp(log(lambda) + lpds);
}
```

When working with only two components there is also a `log_mix` function that incorporates the mixture probability directly.

```
log_mix(theta, foo1_lpdf(x[n]), foo2_lpdf(x[n]))
=
log_sum_exp(log(theta)     + foo1_lpdf(x[n]),
            log(1 - theta) + foo2_lpdf(x[n]))
```

All of this said there is one additional numerical concern relevant to the implementation of mixture models. If any categorical probability vanishes,

$$\lambda_k = 0,$$

then the input to the log sum exp function will overflow to negative infinity,

$$\begin{aligned} v_{k'} &= \log(\lambda_{k'}) + \log \circ p_{k'}(x) \\ &= \log(0) + \log \circ p_{k'}(x) \\ &= -\infty + \log \circ p_{k'}(x) \\ &= -\infty. \end{aligned}$$

At the same time negative infinities also arise if any of the component probability density functions vanish,

$$\begin{aligned} v_{k'} &= \log(\lambda_{k'}) + \log \circ p_{k'}(x) \\ &= \log(\lambda_{k'}) + \log(0) \\ &= \log(\lambda_{k'}) - \infty \\ &= -\infty. \end{aligned}$$

Formally any negative infinite input results in a vanishing exponential output

$$\exp(v_k) = \exp(-\infty) = 0$$

and no contribution to the intermediate sum.

$$\sum_{k=1}^{K} \exp(v_k) = \sum_{k \neq k'} \exp(v_k),$$

Consequently we have, at least in theory, the identity

$$\begin{aligned}\text{log-sum-exp}(v_1, \ldots, v_{k-1}, v_k = -\infty, v_{k+1}, \ldots, v_K) \\ = \text{log-sum-exp}(v_1, \ldots, v_{k-1}, v_{k+1}, \ldots, v_K).\end{aligned}$$

In practice, however, the negative infinity can wreak havoc on the floating point calculations.

To avoid potential numerical complications entirely we need to drop the offending component *before* trying to evaluate the inputs to the log sum exp function. Specifically we compute

$$v_k = \log(\lambda_k) + \log \circ p_k(x)$$

for only the components with $\lambda_k > 0$ and $p_k(x) > 0$, and then evaluate the log sum exp function on only these well-behaved inputs.

1.4 Sampling From Mixture Models

We've done a good bit of work to remove the component assignments from mixture models, but they are not without their uses. Indeed they make sampling from a mixture model particularly straightforward.

In particular the unmarginalized joint model

$$p(x, z \mid \lambda_1, \ldots, \lambda_K) = p(x \mid z)\, p(z \mid \lambda_1, \ldots, \lambda_K)$$

immediately motivates an ancestral sampling strategy for generating samples of x and z. We first select a component by sampling a categorical variable (Figure 1a)

$$\tilde{z} \sim p(z \mid \lambda_1, \ldots, \lambda_K)$$

and then we sample from the corresponding component probability distribution (Figure 1b)

$$\tilde{x} \sim p(x \mid \tilde{z}) = p_{\tilde{z}}(y).$$

Together the pair $\{\tilde{x}, \tilde{z}\}$ defines a sample from the unmarginalized model,

$$\{\tilde{x}, \tilde{z}\} \sim p(x, z \mid \lambda_1, \ldots, \lambda_K),$$

while the single value $\tilde{x}$ defines a sample from the marginalized model,

$$\tilde{x} \sim p(y \mid \lambda_1, \ldots, \lambda_K).$$

Implementing this two-step sampling procedure in the Stan Modeling Language is straightforward.

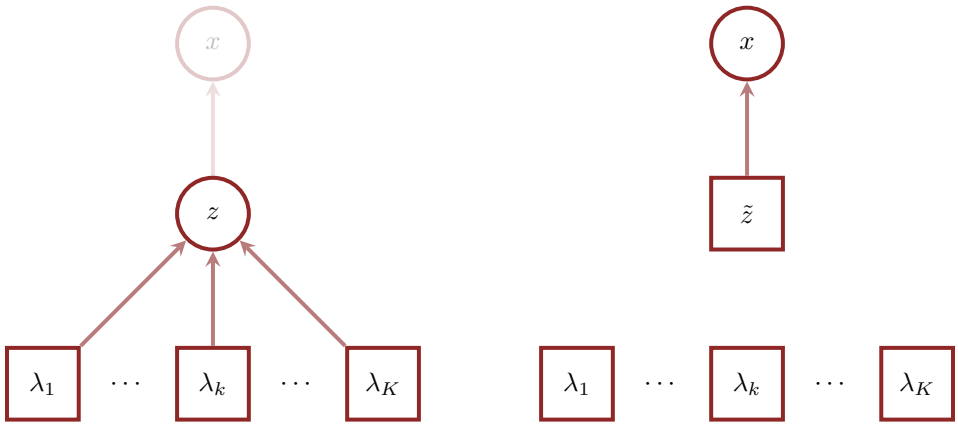


Figure 1: Sampling from a mixture model is a straightforward, two-step process. (a) We first sample a component $\tilde{z}$ given the component probabilities $p(z = k) = \lambda_k$ and then (b) sample a value $\tilde{x}$ from the corresponding component probability distribution $p_{\tilde{z}}(x)$.

```
// Loop over observations
for (n in 1:N) {
  // Sample a component
  int<lower=1, upper=K> z = categorical_rng(lambda);
  // Sample an observation
  if (z == 1) {
    x[n] = foo1_rng();
  } else if (z == 2) {
    x[n] = foo2_rng();
  } else if (z == 3) {
    x[n] = foo3_rng();
  } ...
}
```

Given values for the component probabilities and component probability distributions we can also conditionally sample component assignments for any value using the posterior distribution that we derived in Section 1.1,

$$\tilde{z} \sim \frac{p_z(\tilde{x})\,\lambda_z}{\sum_{k=1}^{K} p_k(\tilde{x})\,\lambda_k}.$$

For example we could sample marginal posterior assignments using the `generated quantities` block in a Stan program.

```
// Loop over observations
for (n in 1:N) {
  vector[4] log_ps =   log(lambda[1]) + foo1_lpdf(x[n])
                     + log(lambda[2]) + foo2_lpdf(x[n])
                     + log(lambda[3]) + foo3_lpdf(x[n])
                     + log(lambda[4]) + foo4_lpdf(x[n]);
  simplex[4] ps = softmax(log_ps);
  z[n] = categorical_rng(ps);
}
```

Equivalently we can use the built-int `categorical_logit_rng` function which automatically applies the `softmax` function to the input arguments.

```
// Loop over observations
for (n in 1:N) {
  vector[4] log_ps =   log(lambda[1]) + foo1_lpdf(x[n])
                     + log(lambda[2]) + foo2_lpdf(x[n])
                     + log(lambda[3]) + foo3_lpdf(x[n])
                     + log(lambda[4]) + foo4_lpdf(x[n]);
  z[n] = categorical_log_rng(log_ps);
}
```

2 Notable Mixture Models

Mixture modeling is a very general modeling technique but there are a few special cases worth particular discussion.

2.1 Inflation Models

Inflation models are a special case of mixture modeling where one or more of the component probability distributions concentrate on single values. These singular probability distributions are defined by the allocations

$$\delta_{x_{\text{inf}}}(\mathsf{x}) = \begin{cases} 1, & x_{\text{inf}} \in \mathsf{x}, \\ 0, & x_{\text{inf}} \notin \mathsf{x} \end{cases}.$$

They are also known as **Dirac probability distributions**, or simply Dirac distributions for short.

While this definition is straightforward, the implementation of inflation models requires some care, especially when the Dirac probability distribution is not compatible with the natural reference measure on the ambient space and we cannot construct well-defined probability density functions.

In theory inflation models can contain any number of component probability distributions but in this section I will consider only two, one baseline component and one inflated component, to simplify the presentation.

2.1.1 Discrete Inflation Models

On discrete spaces any Dirac probability distribution can be specified with a probability density function relative to the counting measure, in other words a probability mass function,

$$\delta_{x_{\text{inf}}}(x) = \begin{cases} 1, & x = x_{\text{inf}}, \\ 0, & x \neq x_{\text{inf}} \end{cases}.$$

To allows us to implement a corresponding inflation model using probability mass functions.

For example we can take a baseline probability distribution specified by the probability mass function $p_{\text{baseline}}(x)$ and inflate the value $x = x_{\text{inf}}$ by mixing it with a Dirac probability mass function,

$$p(x) = \lambda \, p_{\text{baseline}}(x) + (1 - \lambda) \, \delta_{x_{\text{inf}}}(x),$$

or equivalently

$$\begin{aligned} \log \circ p(x_{\text{inf}}) = \text{log-sum-exp}(& \log(\lambda) + \log \circ p_{\text{baseline}}(x_{\text{inf}}), \\ & \log(1 - \lambda) + \log \circ \delta_{x_{\text{inf}}}(x_{\text{inf}}) \ ). \end{aligned}$$

Because the Dirac component allocates zero probability to so many elements, however, we have to take care when evaluating the logarithm of this mixed probability mass function. If $x = x_{\text{inf}}$ then

$$\delta_{x_{\text{inf}}}(x = x_{\text{inf}}) = 1$$

and

$$\log \circ \delta_{x_{\text{inf}}}(x = x_{\text{inf}}) = 0.$$

At this point evaluating the log probability mass function is straightforward,

$$\begin{aligned} \log \circ p(x_{\text{inf}}) &= \text{log-sum-exp}(\log(\lambda) + \log \circ p_{\text{baseline}}(x_{\text{inf}}), \\ &\qquad \log(1 - \lambda) + \log \circ \delta_{x_{\text{inf}}}(x_{\text{inf}}) \ ) \\ &= \text{log-sum-exp}(\log(\lambda) + \log \circ p_{\text{baseline}}(x_{\text{inf}})), \\ &\qquad \log(1 - \lambda) + \log(1) \ ) \\ &= \text{log-sum-exp}(\log(\lambda) + \log \circ p_{\text{baseline}}(x_{\text{inf}})), \\ &\qquad \log(1 - \lambda) \ ). \end{aligned}$$

On the other hand if $x \neq x_{\text{inf}}$ then

$$\delta_{x_{\text{inf}}}(x \neq x_{\text{inf}}) = 0$$

and

$$\log \circ \delta_{x_{\text{inf}}}(x \neq x_{\text{inf}}) = -\infty.$$

To avoid numerical issues arising from the negative infinity we need to follow the procedure introduced in Section 1.3, first analytically accounting for the zero,

$$\begin{aligned} p(x) &= \lambda \, p_{\text{baseline}}(x) + (1 - \lambda) \, \delta_{x_{\text{inf}}}(x) \\ &= \lambda \, p_{\text{baseline}}(x) + (1 - \lambda) \, 0 \\ &= \lambda \, p_{\text{baseline}}(x), \end{aligned}$$

and only then taking the natural logarithm,

$$\begin{aligned} \log \circ p(y) &= \log \left(\lambda \, p_{\text{baseline}}(x) \right) \\ &= \log(\lambda) + \log \circ p_{\text{baseline}}(x). \end{aligned}$$

In other words to ensure stable numerical calculations in practice we have to invoke conditional statements when implementing the discrete inflation model. For example in Stan we might write

```
// Loop over observations
for (n in 1:N) {
  // Check equality with inflated value
  if (x[n] == x_inf) {
    target += log_sum_exp(log(lambda) + log_p_baseline(x[n]),
                          log(1 - lambda));
  } else {
    target += log(lambda) + log_p_baseline(x[n]);
  }
}
```

2.1.2 Continuous Inflation Models

Unfortunately Dirac probability distributions are less well-behaved on continuous spaces. The problem is that Dirac probability distributions are not absolutely continuous with respect to the natural reference measures, such as the Lebesgue measure on a given real space. Consequently we cannot define an appropriate probability density function.

A common heuristic for denoting a continuous inflation model uses the Dirac delta function $\delta(x)$. Recall that the Dirac delta function is defined by the integral action

$$\int \mathrm{d}x \, \delta(x) f(x) = f(0)$$

and doesn't actually have a well-defined point-wise output.

Using a Dirac delta function we can heuristically write a continuous inflation model with a single inflated value x_{inf} as

$$p(x) = \lambda \, p_{\text{baseline}}(x) + (1 - \lambda) \, \delta(x - x_{\text{inf}})$$

but, because we cannot evaluate $\delta(x - x_{\text{inf}})$, we actually cannot evaluate $p(x)$.

In order to formally implement a continuous inflation model we need to break the ambient space X into two pieces, once containing the inflated value x_{inf} and one containing everything else $X \setminus x_{\text{inf}}$. We can then treat $\{x_{\text{inf}}\}$ as a discrete space equipped with a counting reference measure and and $X \setminus x_{\text{inf}}$ as a continuous space equipped with, for example, a Lebesgue reference measure.

If π_{baseline} is absolutely continuous with respect to a Lebesgue reference measure on X then the probability allocated to any singular inflated value will be zero

$$\pi_{\text{baseline}}(\{x_{\text{inf}}\}) = 0.$$

Consequently the baseline component will effectively define the same probability distribution on X and $X \setminus x_{\text{inf}}$, and we can represent both with the same probability density function. In other words for $x \neq x_{\text{inf}}$ the inflation model can be specified by the probability density function

$$p(x) = \lambda \, p_{\text{baseline}}(x).$$

On the other hand for $x = x_{\text{inf}}$ we have to treat the mixture probability density function as a probability mass function,

$$\begin{aligned} p(x_{\text{inf}}) &= \pi(\{x_{\text{inf}}\}) \\ &= \lambda \, \pi_{\text{baseline}}(\{x_{\text{inf}}\}) + (1 - \lambda) \, \delta_{x_{\text{inf}}}(\{x_{\text{inf}}\}) \\ &= \lambda \, 0 + (1 - \lambda) \, 1 \\ &= (1 - \lambda). \end{aligned}$$

When working with N independent instances it's convenient to first separate the values into the N_{inf} instances that exactly equal the inflated value,

$$v_1, \ldots, v_{N_{\text{inf}}},$$

and the remaining $N - N_{\text{inf}}$ observations that don't,

$$w_1, \ldots, w_{N - N_{\text{inf}}}.$$

This allows us to simplify the joint probability density function into the form

$$\begin{aligned}
p(x_1, \ldots, x_N) &= \prod_{n=1}^{N} p(x_n) \\
&= \prod_{n=1}^{N_{\text{inf}}} p(v_n) \prod_{n=1}^{N - N_{\text{inf}}} p(w_n) \\
&= \prod_{n=1}^{N_{\text{inf}}} (1 - \lambda) \prod_{n=1}^{N - N_{\text{inf}}} \lambda \, p_{\text{baseline}}(w_n) \\
&= (1 - \lambda)^{N_{\text{inf}}} \, \lambda^{N - N_{\text{inf}}} \prod_{n=1}^{N - N_{\text{inf}}} p_{\text{baseline}}(w_n) \\
&\propto \text{Binomial}(N_{\text{inf}} \mid N, 1 - \lambda) \prod_{n=1}^{N - N_{\text{inf}}} p_{\text{baseline}}(w_n).
\end{aligned}$$

In other words the continuous inflation model completely decouples into a Binomial model for the number of inflated values and a baseline model for the non-inflated values! Moreover, because these models are independent of each other they can be implemented together or separately *without impacting any inferences*.

The intuition here is that when evaluating the mixture model on the inflated value the contribution of the inflating component is always infinitely larger than the contribution from any other components. If we encounter x_{inf} then we know that it *has* to be associated with the inflated component, and if we observe any other value then we know that it *has* to be associated with another component. Because of this lack of ambiguity we can always separate the inflated and non-inflated values and model them separately without compromising the consistency of our inferences.

2.2 Categorical and Multinomial Mixture Models

Categorical probability distributions are relatively simple objects, but mixtures of categorical probability distributions exhibit a unique property that can be of use in some circumstances.

Recall that every categorical probability distribution over a set of K unstructured elements is defined by K atomic probabilities

$$\text{categorical}(\{k\} \mid \mathbf{q}) = q_k.$$

Conveniently we can also write this as a mixture model where every component probability distribution is a Dirac probability distribution,

$$\text{categorical}(\mathsf{z} \mid \mathbf{q}) = \sum_{k=1}^{K} q_k \, \delta_{\{k\}}(\mathsf{z}),$$

or equivalently a mixture probability mass function,

$$\text{categorical}(z \mid \mathbf{q}) = \sum_{k=1}^{K} q_k \, \delta_k(z),$$

where the Dirac probability distribution and probability mass function behave as in Section 2.1.1.

Interestingly the mixture of two different categorical probability distributions is always another categorical probability distribution,

$$\begin{aligned} &\lambda \, \text{categorical}(z \mid \mathbf{q}_1) \\ +(1 - \lambda) \, \text{categorical}(z \mid \mathbf{q}_2) &= \lambda \sum_{k=1}^{K} q_{1k} \, \delta_k(z) + (1 - \lambda) \sum_{k=1}^{K} q_{2k} \, \delta_k(z) \\ &= \sum_{k=1}^{K} \left[\lambda \, q_{1k} + (1 - \lambda) \, q_{2k} \right] \, \delta_k(z) \\ &= \text{categorical}(z \mid \lambda \, \mathbf{q}_1 + (1 - \lambda) \, \mathbf{q}_2). \end{aligned}$$

Indeed the mixture of any number of categorical probability distributions defines another categorical probability distribution,

$$\begin{aligned} \sum_{j=1}^{J} \lambda_j \text{categorical}(z \mid \mathbf{q}_j) &= \sum_{j=1}^{J} \lambda_j \sum_{k=1}^{K} q_{jk} \, \delta_k(z) \\ &= \sum_{k=1}^{K} \left[\sum_{j=1}^{J} \lambda_j \, q_{jk} \right] \, \delta_k(z) \\ &= \text{categorical}(z \mid \textstyle\sum_{j=1}^{J} \lambda_j \, \mathbf{q}_j). \end{aligned}$$

In other words mixing categorical probability distributions is equivalent to mixing the corresponding simplex configurations together.

That said the closure of categorical probability distributions under mixtures, and the corresponding duality between probability distribution mixtures and simplex configuration mixtures, is exceptional. Consider, for example, the multinomial probability distribution over the total counts over an ensemble of identical and independent categorical instances,

$$\text{multinomial}(n_1, \ldots, n_K \mid \mathbf{q}) = N! \prod_{k=1}^{K} \frac{q_k^{n_k}}{n_k!}.$$

While mixing simplex configurations together is equivalent to mixing the corresponding categorical probability distributions together,

$$\begin{aligned} &\text{categorical}(z \mid \textstyle\sum_{j=1}^{J} \lambda_j \, \mathbf{q}_j) \\ &\quad = \sum_{j=1}^{J} \lambda_j \text{categorical}(z \mid \mathbf{q}_j), \end{aligned}$$

it is not equivalent to mixing the corresponding multinomial probability distributions together (Figure 2)

$$\begin{aligned} &\text{multinomial}(n_1, \ldots, n_K \mid \textstyle\sum_{j=1}^{J} \lambda_j \, \mathbf{q}_j) \\ &\quad \neq \sum_{j=1}^{J} \lambda_j \text{multinomial}(n_1, \ldots, n_K \mid \mathbf{q}_j)! \end{aligned}$$

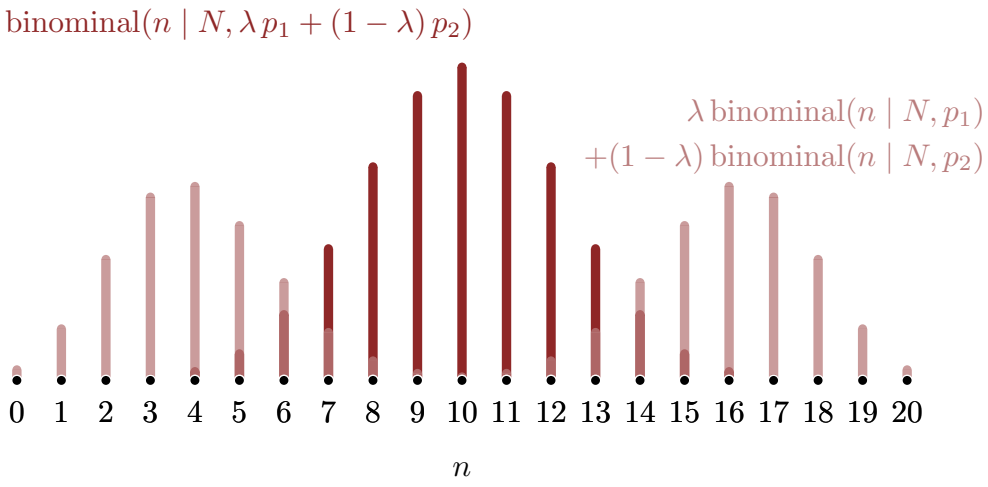


Figure 2: Mixing categorical probability distributions is equivalent to mixing the defining simplex configurations, but this property does not generalize to other probability distributions derived from categorical probability distributions, such as the multinomial distribution. For example a mixture of the two binomial probability distributions $\text{binominal}(n \mid N, p_1)$ and $\text{binominal}(n \mid N, p_2)$ is not the same as the binomial distribution defined by the mixture of p_1 and p_2 even though $\lambda \, \text{Bernoulli}(p_1) + (1 - \lambda) \, \text{Bernoulli}(p_2)$ is the same as $\text{Bernoulli}(\lambda \, p_1 + (1 - \lambda) \, p_2)$.

The left-hand side

$$\text{multinomial}(n_1, \ldots, n_K \mid \textstyle\sum_{j=1}^{J} \lambda_j \, \mathbf{q}_j)$$

corresponds to *heterogeneous* component categorical probability distributions. Specifically it implies that a proportion λ_j of the N component categorical probability distributions are

configured by $\mathbf{q}_j$. These heterogeneous contributions to the counts wash out the detail of each component model, always resulting in a uni-modal probability distribution.

On the other side

$$\sum_{j=1}^{J} \lambda_j \text{multinomial}(n_1, \ldots, n_K \mid \mathbf{q}_j)$$

corresponds to *homogeneous* component categorical probability distributions. For each simplex configuration $\mathbf{q}_j$ the corresponding multinomial distribution concentrates around the count values $q_{jk}\, N$. The mixture model overlays these distinct peaks on top of each other, resulting in a multi-modal probability distribution.

In general we have to take care to avoid misinterpreting models defined by mixtures of parameters as probabilistic mixture models.

2.3 Continuous Mixture Models

So far we have considered mixture models with only a finite number of components. These models are defined by the joint probability density function

$$p(x, z) = p(x \mid z)\, p(z)$$

with

$$z \in Z = \{1, \ldots, k, \ldots, K\},$$

or equivalently the marginal probability density function

$$\begin{aligned} p(x) &= \sum_{k=1}^{K} p(x \mid k)\, p(k) \\ &= \sum_{k=1}^{K} p_k(x)\, \lambda_k \\ &= \sum_{k=1}^{K} \lambda_k\, p_k(x). \end{aligned}$$

We can immediately extend this construction to mixture models with a countably infinite number of components,

$$p(x, z) = p(x \mid z)\, p(z)$$

with

$$z \in Z = \{1, \ldots, k, \ldots, \},$$

or equivalently

$$\begin{aligned} p(x) &= \sum_{k=1}^{\infty} p(x \mid k)\, p(k) \\ &= \sum_{k=1}^{\infty} p_k(x)\, \lambda_k \\ &= \sum_{k=1}^{\infty} \lambda_k\, p_k(x). \end{aligned}$$

These models are referred to as **discrete mixture models** because the categorical assignment variables take values in discrete spaces. To be more precise they can also be referred to as finite discrete mixture models and infinite discrete mixture models, respectively.

Mathematics, however, allows us to take this construction even further: by allowing the assignment variable to take values in an arbitrary space we can build mixture models from an arbitrary number of components. For example if $Z = \mathbb{R}$ then the joint model

$$p(x, z) = p(x \mid z)\, p(z).$$

can be interpreted as a mixture model with an uncountably infinite number of component models, each of which is specified by the conditional probability density function $p(x \mid z)$ and weighted by not a probability but rather a probability density $p(z)$.

In this case the marginal mixture model is defined not by summation but rather integration,

$$\begin{aligned} p(x) &= \int \mathrm{d}z\, p(z)\, p(x \mid z) \\ &= \int \mathrm{d}z\, p(x \mid z)\, p(z). \end{aligned}$$

Because the assignment variables now take values in a continuous space mixture models of this form are, unsurprisingly, referred to as **continuous mixture models**.

From a practical perspective continuous mixture models are a bit of a double-edged sword. On one hand the integration needed to evaluate the marginal model $p(x)$ can be implemented analytically only in exceptional circumstances. On the other hand if X is also a continuous space then the joint model $p(x, z)$ will be defined over a completely continuous product space and we can use gradient-based tools like Hamiltonian Monte Carlo to characterize it directly.

3 Bayesian Mixture Models

When working with probabilistic models defined over products spaces we have a variety of ways that we can employ mixture modeling. For example we can use mixtures to model the

behavior of the entire product space. At the same time we can use mixtures to model the behavior of individual components or even products of specific components.

This flexibility becomes especially useful in Bayesian modeling where our models are defined over a product of an observational space and a model configuration space. In particular we can apply mixture modeling techniques to both the prior model $p(\theta)$ and observational model $p(y \mid \theta)$.

3.1 Mixture Prior Models

Applied prior modeling is often constrained by our ability to express our intricate domain expertise with the mathematically-convenient, but relatively simple, probability density functions to which we might be limited in practice. That said when we don't have access to sufficiently flexible families of probability density functions we can often engineer some of the desired flexibility by combining the available probability density functions into mixture prior models,

$$p(\theta) = \sum_{k=1}^{K} \lambda_k \, p_k(\theta).$$

For example let's say that we want to build a principled prior model for a behavior modeled by the parameter θ, but we have access to only normal probability density functions. In this case a unimodal normal prior model is sufficient only if we want to suppress parameter configurations outside of a single interval (Figure 3).

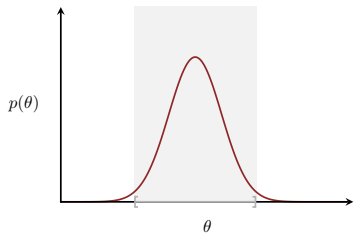
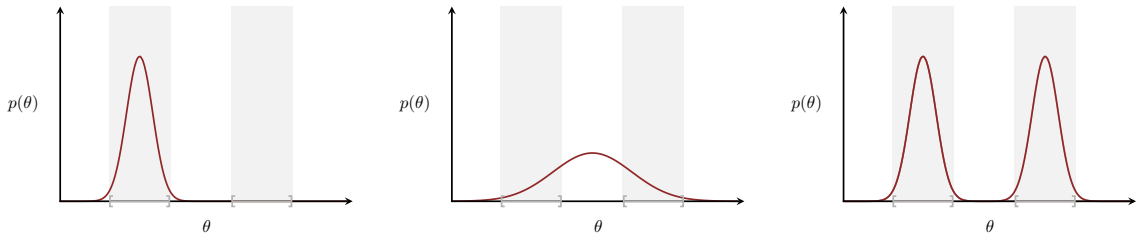


Figure 3: A single normal prior model is useful for quantifying domain expertise that is consistent with a connected interval of parameter configurations.

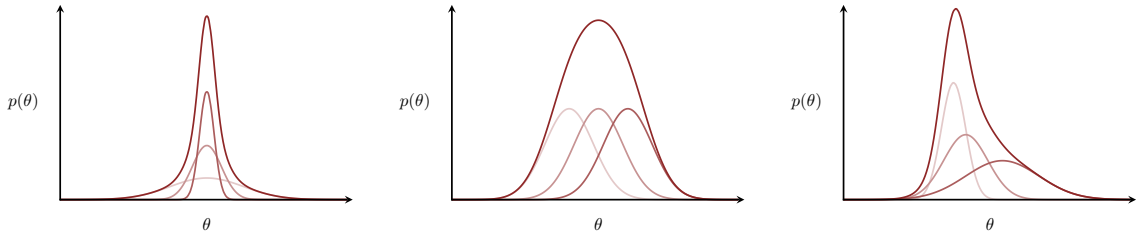
If our domain expertise is consistent with parameter configurations that concentrate within multiple disjoint intervals, however, then no single normal prior model will be sufficient. We can build a normal prior model that captures any one interval, but only at the expense of the others (Figure 4a). At the same time a normal prior model that expands to contain *all* of the consistent intervals will end up including the inconsistent behaviors in between (Figure 4b). On the other hand a mixture of normal prior models can accommodate the disjoint intervals without issue (Figure 4c).



(a) (b) (c)

Figure 4: A single normal prior model is less useful for quantifying domain expertise that is consistent with multiple, disconnected intervals of parameter configurations. The best we can do is engineer a normal prior model (a) that concentrates on one of the intervals but neglects the others (b) or spans all of the intervals but includes all of the inconsistent configurations between them. (c) Multiple normal prior models, however, can be combined into a mixture prior model that readily accommodates the disjoint domain expertise.

Mixture models can also be useful for engineering more sophisticated unimodal behaviors. For example we can use a mixture of normal prior models to approximate more leptokurtic or "heavy-tailed" behaviors (Figure 5a), platykurtic or "light-tailed" behaviors (Figure 5b), and even asymmetric behaviors (Figure 5c).



(a) (b) (c)

Figure 5: Mixture prior models can be used to engineer more sophisticated behavior from simpler component models. For example a mixture of normal prior models can be more (a) heavy-tailed, (b) light-tailed, or (c) asymmetric than any single normal prior model.

All of this said, multimodal domain expertise is not particularly common in practice. Unimodal domain expertise that cannot be captured by a normal prior model is more common but, with the benefit of expressive probabilistic programming languages and contemporary probabilistic computation tools, we can usually implement sufficiently flexible probability density functions directly. Prior mixture modeling was much more useful in times past when Bayesian analyses were limited to calculations that could be made analytically or with limited computational

resources.

Continuous mixture prior models, however, still play a critical role in the modeling of domain expertise subject to certain ignorance constraints, as we'll learn about in the chapter on hierarchical modeling.

3.2 Mixture Observational Models

One of the most powerful uses of mixtures models in Bayesian analyses is engineering observational models that combine multiple data generating processes together,

$$p(y \mid \theta) = \sum_{k=1}^{K} \lambda_k \, p_k(y \mid \theta),$$

when we don't know which might be responsible for any given observation. These mixture observational models allow us to, for example, model measurements that have been contaminated by undesired but unavoidable behaviors, a situation that arises all too often in practice.

For example if $p_s(y \mid \theta_s)$ models a desired signal and $p_b(y \mid \theta_b)$ models an irreducible background then the mixture observational model

$$p(y \mid \theta_s, \theta_b) = \lambda_s \, p_s(y \mid \theta_s) + (1 - \lambda_s) \, p_b(y \mid \theta_b)$$

models their overlap, allowing us to *probabilistically* separate the signal from the background.

Similarly consider a normal observational model $\text{normal}(y \mid, \mu, \sigma)$ where the measurement variability σ can vary unpredictably from measurement to measurement. If we can quantify the distribution of measurement variabilities then we can model this data generating behavior with a continuous mixture model, for example

$$\begin{aligned} p(y \mid \mu, \tau) &= \int_0^{\infty} \mathrm{d}z \, \text{normal}\left(y \mid \mu, \sqrt{z}\right) \text{exponential}\left(z \mid \frac{1}{2\tau^2}\right) \\ &= \text{Laplace}(y \mid \mu, \tau). \end{aligned}$$

While mathematically convenient, taking exponentially distributed squared measurement variabilities is a strong assumption that we usually don't have the domain expertise to confidently motivate. Instead in practice this Laplace observational model is often used more heuristically to accommodate extreme "outliers" that contaminate the real-valued observations of interest.

4 Mixture Observational Model Inferences

In practice mixture observational models are typically used in applications where the neither the component observational models nor their probabilities are known precisely. Consequently the inferential challenge is to infer all of these behaviors from the observed data *at the same time*.

Theoretically we can always infer the component observational model behaviors jointly with the unknown component assignments,

$$\begin{aligned} p(\mathbf{z}, \lambda, \theta \mid \tilde{\mathbf{y}}) &\propto p(\tilde{\mathbf{y}}, \mathbf{z}, \lambda, \theta) \\ &\propto \left[p(\tilde{\mathbf{y}} \mid \mathbf{z}, \theta) \, p(\mathbf{z} \mid \lambda) \right] p(\lambda) \, p(\theta) \\ &\propto \prod_{n=1}^{N} \left[p_{z_n}(\tilde{y}_n \mid \theta_{z_n}) \, \lambda_{z_n} \right] p(\lambda) \, p(\theta). \end{aligned}$$

That said in practice it is almost always easier to infer the component probabilities and component model configurations directly from the marginalized model,

$$\begin{aligned} p(\lambda, \theta \mid \tilde{\mathbf{y}}) &\propto p(\tilde{\mathbf{y}}, \lambda, \theta) \\ &\propto \left[p(\tilde{\mathbf{y}} \mid \lambda, \theta) \right] p(\lambda) \, p(\theta) \\ &\propto \prod_{n=1}^{N} \left[\sum_{k=1}^{K} p_k(\tilde{y}_n \mid \theta_k) \, \lambda_k \right] p(\lambda) \, p(\theta). \end{aligned}$$

When needed any component assignment can always be recovered with an application of Bayes' Theorem,

$$p(z_n = k \mid \tilde{y}_n, \lambda, \theta) = \frac{p_k(\tilde{y}_n \mid \theta_k) \, \lambda_k}{\sum_{k'=1}^{K} p_{k'}(\tilde{y}_n \mid \theta_k) \, \lambda_k}.$$

From a mathematical perspective the behavior of mixture observational model inferences is relatively straightforward. For example if the configurations of the component observational models are fixed then any observations that are more consistent with a particular component model,

$$\frac{\mathrm{d}\pi_{k, \theta_k}}{\mathrm{d}\pi_{k', \theta_{k'}}}(\tilde{y}_n) = \frac{p_k(\tilde{y}_n \mid \theta_k)}{p_{k'}(\tilde{y}_n \mid \theta_{k'})} > 1$$

for all $k' \neq k$, will push the posterior distribution to concentrate on larger values of λ_k and smaller values of the remaining $\lambda_{k'}$.

More generally the posterior distribution will have to account for the interactions between the component probabilities λ and the component model configuration parameters θ. When

the component observational models are not too redundant, with each component model responsible for a mostly unique set of behaviors, then then these interactions will manifest as non-degenerate posterior distribution that are straightforward to quantify with tools like Hamiltonian Monte Carlo.

If the component observational models exhibit substantial redundancy, however, then the resulting posterior inferences will be much less pleasant. The problem is that the mixture observational model will be able to contort itself in a variety of different ways, all of which are consistent with the observed data. This yields complex, and often multi-modal, degeneracies that frustrate accurate posterior computation. Recall that multi-modal degeneracies are particularly obstructive as Markov chains with poorly chosen initializations can be trapped by even small modes (Figure 6).

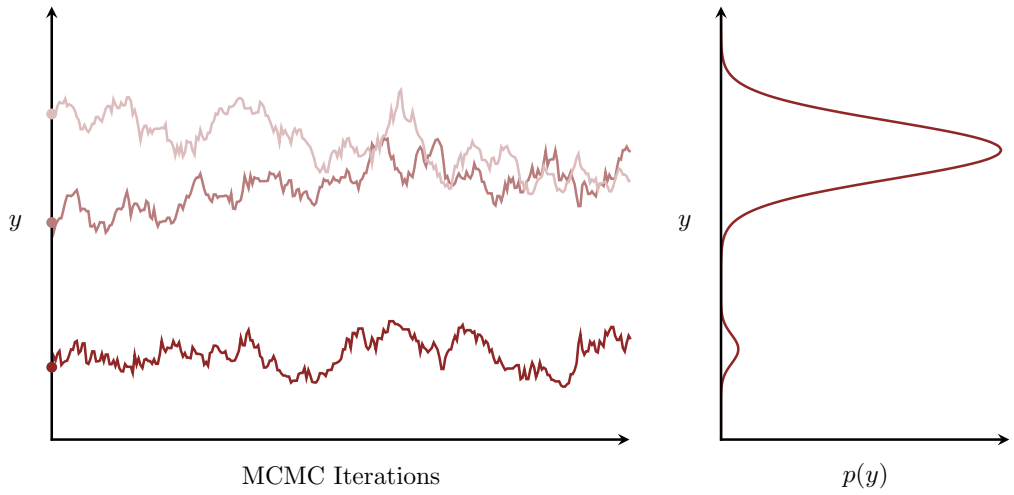


Figure 6: Multi-modal posterior distributions are especially frustrating for Markov chain Monte Carlo. No matter how little target probability is allocated to the neighborhood of a mode it can still attract Markov chains for arbitrarily long times if the Markov chains are initialized too close to its basin of attraction.

Redundancy in a mixture observational model is at its highest when two or more of the component observational models are the same. In this case we can permute the corresponding component indices *without affecting any of the model evaluations*. If K_r component models are the same then the likelihood function will always exhibit $K_r!$ distinct modes, one for each permutation of the redundant indices (Figure 7).

That said non-redundant mixture observational models are not guaranteed to be well-behaved either. Degeneracies can also arise when component observational models are mathematically distinct but contain similar qualitative behaviors.

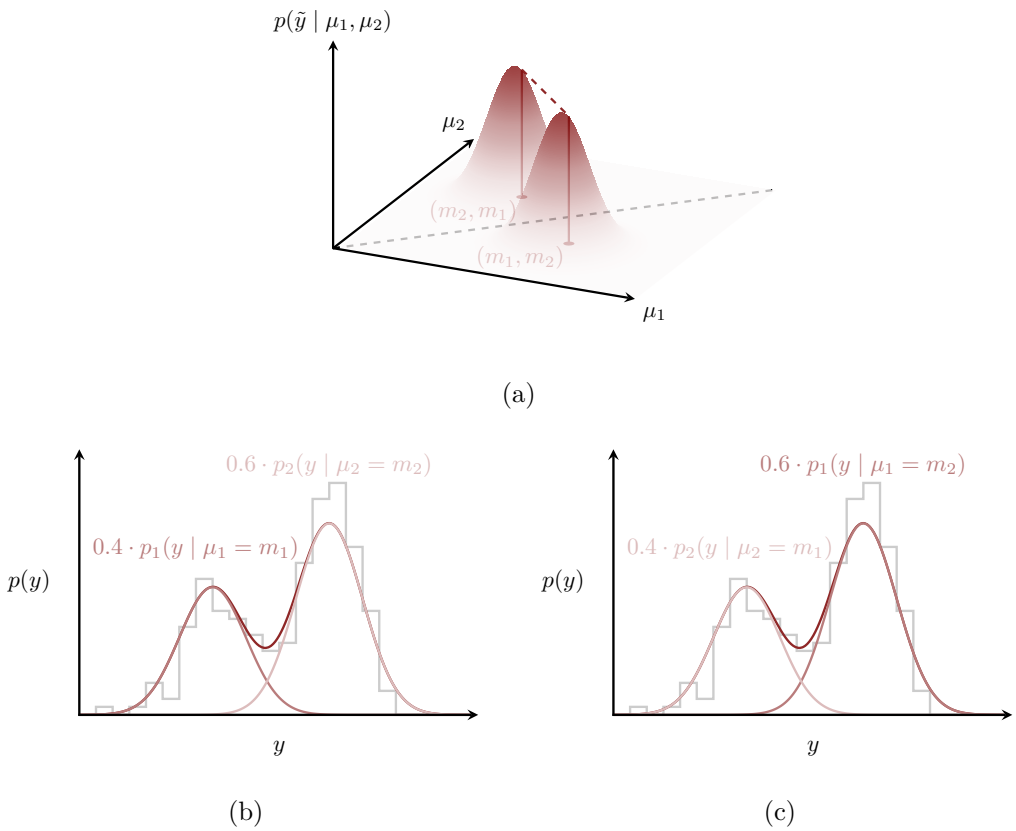
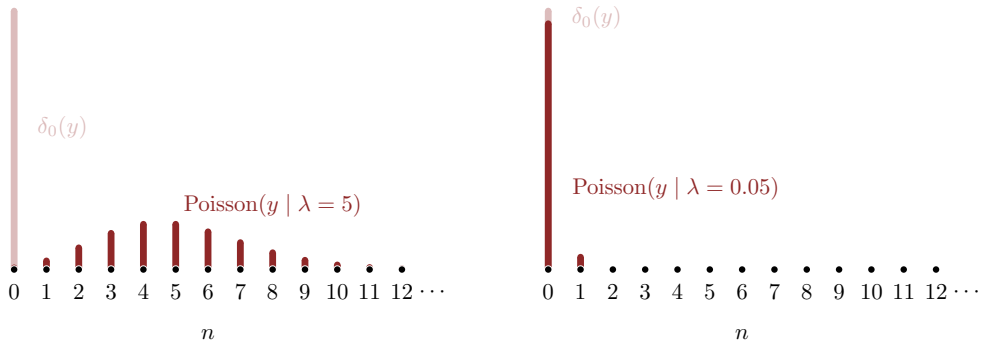


Figure 7: Mixture observational models with K equal component observational models are formally non-identified – every model configuration is accompanied by $K! - 1$ other model configurations that define exactly the same observational model. These redundant model configurations manifest as distinct modes in the likelihood function; for example a mixture observational model with two identical normal component models always results in (a) a bimodal likelihood function where (b) every model configuration in one mode is paired with (c) a model configuration in another mode where the component models are swapped.

Consider, for example, a mixture observational model over positive integers that inflates a baseline Poisson model, $\text{Poisson}(y \mid \lambda)$, with a Dirac probability distribution concentrating at zero, $\delta_0(y)$.

If λ is far from zero then these two component model capture distinct behaviors, with the former concentrating on values above zero and the latter on values exactly at zero (Figure 8a). When λ is close to zero, however, the two component models will both concentrate at zero (Figure 8b) and observations of $y = 0$ will be able to distinguish between them.



(a) (b)

Figure 8: Discrete inflation observational models are prone to degeneracies when the baseline model can contort itself to mimic the inflation model. (a) When λ is much larger than zero a Poisson model is not easily confused with a Dirac model concentrating at zero. (b) If λ is close to zero, however, then the two component observational models are nearly indistinguishable.

In my opinion the most productive approach to mixture observational modeling is to treat each component as a model for a distinct data generating process. Any domain expertise that distinguishes the possible data generating behaviors from each other can, and should, be directly incorporated into either the structure of the component observational models or the prior model to avoid as much inferential degeneracy as possible.

5 Demonstrations

Enough of the generality. In this section we'll demonstrate the basic mixture modeling techniques presented in this chapter with a sequence of demonstrative Bayesian applications using mixture observational models.

5.1 Setup

Always we start by setting up our local environment.

```
par(family="serif", las=1, bty="l",
    cex.axis=1, cex.lab=1, cex.main=1,
    xaxs="i", yaxs="i", mar = c(5, 5, 3, 1))

library(rstan)
rstan_options(auto_write = TRUE)            # Cache compiled Stan programs
options(mc.cores = parallel::detectCores()) # Parallelize chains
parallel:::setDefaultClusterOptions(setup_strategy = "sequential")
```

```
util <- new.env()
source('mcmc_analysis_tools_rstan.R', local=util)
source('mcmc_visualization_tools.R', local=util)
```

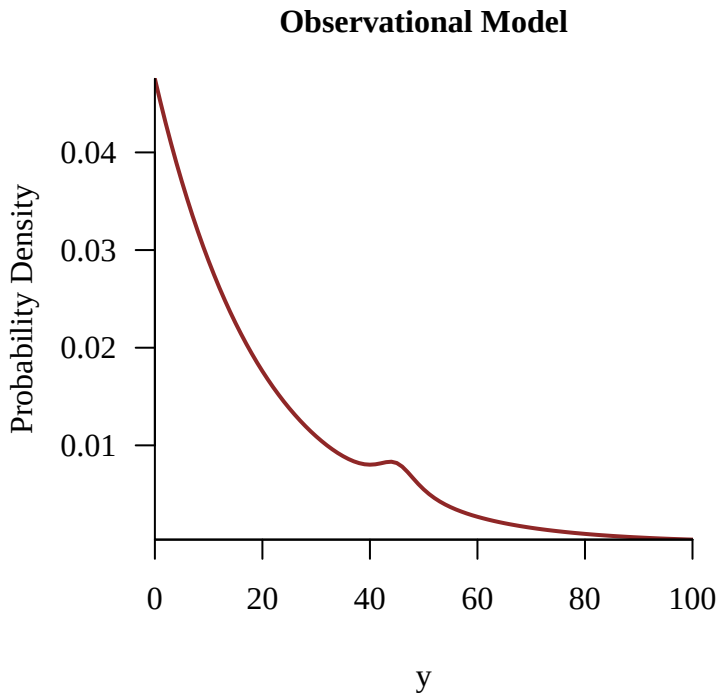
5.2 Separating Signal and Background

For our first exercise let's consider a simplified version of a particle physics experiment where we observe individual energy depositions in a detector. The only problem is that each depositions can come from either an irreducible background or a signal of interest, and we are always ignorant to which source is responsible for any given deposition. Fortunately we can model this overlap of signal and background sources with a mixture observational model.

We'll model the background with a steeply falling exponential probability density function and the signal with a Cauchy probability density function that can be derived as an approximation to certain physical processes. The precise configuration of the signal and background models in this example is somewhat arbitrary, especially without units. That said the high background probability and peaked signal is emulative of actual particle physics experiments.

```
mu_signal <- 45
sigma_signal <- 5
beta_back <- 20
lambda <- 0.95

xs <- seq(0, 100, 1)
ys <- lambda * dexp(xs, 1 / beta_back) +
      (1 - lambda) * dcauchy(xs, mu_signal, sigma_signal)

par(mfrow=c(1, 1), mar=c(5, 5, 3, 1))
```

```
plot(xs, ys, lwd=2, type="l", col=util$c_dark,
     main="Observational Model",
     xlab="y", ylab="Probability Density")
```



Simulating data from this model is straightforward using the techniques introduced in Section 1.4.

```
N <- 500

simu <- stan(file="stan_programs/simu_signal_background.stan",
             algorithm="Fixed_param",
             data=list("N" = N), seed=8438338,
             warmup=0, iter=1, chains=1, refresh=0)

data <- list("N" = N,
             "y" = extract(simu)$y[1,])
```

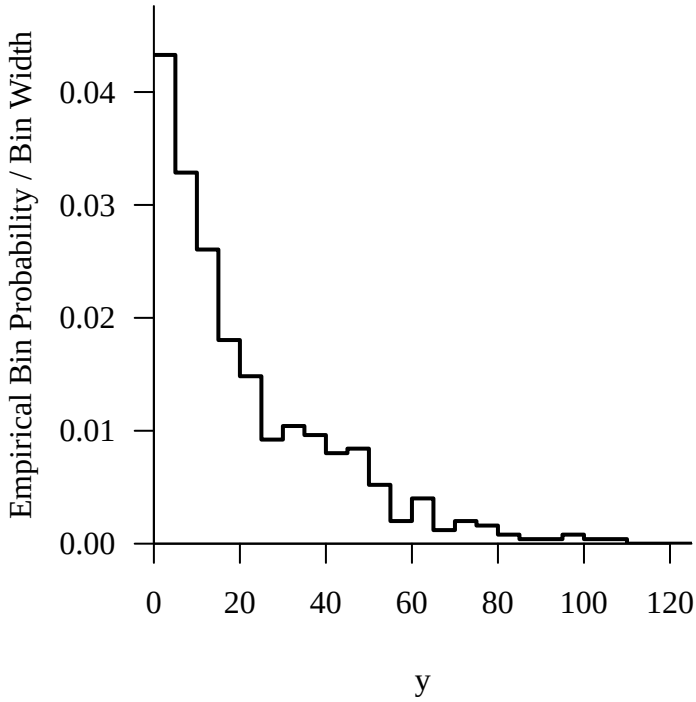
Because of the overwhelming background contribution it's hard to make out the meager signal by eye.

Stan
Program 1 `simu\_signal\_background.stan`

```
data {
  int<lower=1> N;      // Number of observations
}

transformed data {
  real mu_signal =  45;                 // Signal location
  real<lower=0> sigma_signal = 5;       // Signal scale
  real beta_back = 20;                  // Background rate
  real<lower=0, upper=1> lambda = 0.95; // Background probability
}

generated quantities {
  array[N] real<lower=0> y = rep_array(-1, N);

  for (n in 1:N) {
    if (bernoulli_rng(lambda)) {
      y[n] = exponential_rng(1 / beta_back);
    } else {
      // Truncate signal to positive values
      while (y[n] < 0) {
        y[n] = cauchy_rng(mu_signal, sigma_signal);
      }
    }
  }
}
```

```
par(mfrow=c(1, 1), mar=c(5, 5, 1, 1))

util$plot_line_hist(data$y, 0, 125, 5, xlab="y", prob=TRUE)
```

```
Warning in check_bin_containment(bin_min, bin_max, values): 1 value (0.2%) fell
above the binning.
```



Perhaps we can use Bayesian inference to tease out the signal?

Note that the prior model here is a bit arbitrary due to the nature of the exercise. In a practical analysis we would want to take care to elicit at least some basic domain expertise about the behaviors of each component observational model. This is especially true if the component models have any potential for redundant behaviors.

```
fit <- stan(file="stan_programs/signal_background1.stan",
            data=data, seed=8438338,
            warmup=1000, iter=2024, refresh=0)
```

The diagnostics exhibit some mild $\hat{\xi}$ warnings, but nothing too concerning so long as we're not interested in calculating the expectation value of `mu_signal`.

```
diagnostics <- util$extract_hmc_diagnostics(fit)
util$check_all_hmc_diagnostics(diagnostics)
```

```
  All Hamiltonian Monte Carlo diagnostics are consistent with reliable
Markov chain Monte Carlo.
```

```
samples <- util$extract_expectand_vals(fit)
base_samples <- util$filter_expectands(samples,
                                       c('mu_signal', 'sigma_signal',
                                         'beta_back', 'lambda'))
util$check_all_expectand_diagnostics(base_samples)
```

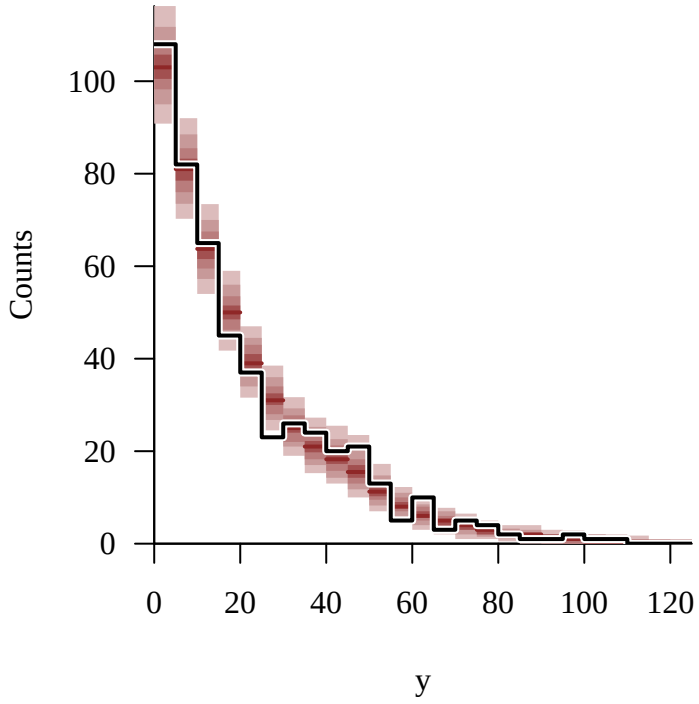
```
mu_signal:
  Chain 1: Right tail hat{xi} (0.253) exceeds 0.25.
  Chain 3: Right tail hat{xi} (0.423) exceeds 0.25.
  Chain 4: Right tail hat{xi} (0.378) exceeds 0.25.


Large tail hat{xi}s suggest that the expectand might not be
sufficiently integrable.
```

Even better there are no indications of retrodictive tension that would suggest inadequacies in our model.

```
par(mfrow=c(1, 1), mar=c(5, 5, 1, 1))

util$plot_hist_quantiles(samples, 'y_pred', 0, 125, 5,
                         baseline_values=data$y, xlab="y")
```

```
Warning in check_bin_containment(bin_min, bin_max, collapsed_values,
"predictive value"): 6981 predictive values (0.3%) fell above the binning.
```

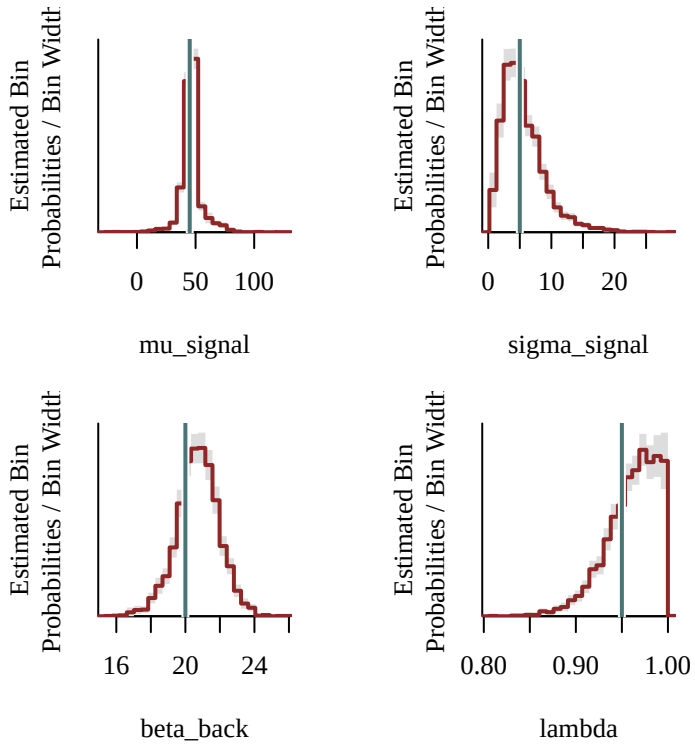
```
Warning in check_bin_containment(bin_min, bin_max, baseline_values, "observed
value"): 1 observed value (0.2%) fell above the binning.
```



Our posterior inferences are not too bad, especially considering the relatively small number of observations and overwhelming background contribution. Note also the heavy tails of the marginal posterior distribution for `mu_signal`, consistent with the $\hat{\xi}$ warnings.

```
par(mfrow=c(2, 2), mar=c(5, 5, 1, 1))

util$plot_expectand_pushforward(samples[['mu_signal']], 25,
                                display_name="mu_signal",
                                baseline=mu_signal,
                                baseline_col=util$c_mid_teal)

util$plot_expectand_pushforward(samples[['sigma_signal']], 25,
                                display_name="sigma_signal",
                                baseline=sigma_signal,
                                baseline_col=util$c_mid_teal)

util$plot_expectand_pushforward(samples[['beta_back']], 25,
                                display_name="beta_back",
                                baseline=beta_back,
                                baseline_col=util$c_mid_teal)

util$plot_expectand_pushforward(samples[['lambda']], 25,
                                display_name="lambda",
```

```
                                baseline=lambda,
                                baseline_col=util$c_mid_teal)
```



We can also directly visualize the inferred behaviors of the signal model, the background model, and their combinations.

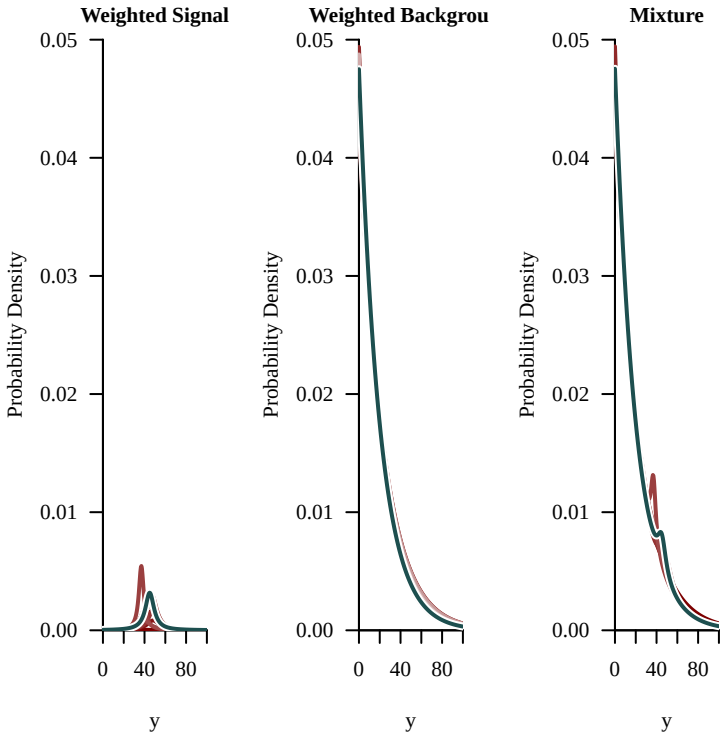
```
library(colormap)
nom_colors <- c("#DCBCBC", "#C79999", "#B97C7C",
                "#A25050", "#8F2727", "#7C0000")
line_colors <- colormap(colormap=nom_colors, nshades=20)

cs <- c(1, 2, 3, 4)
ss <- c(1, 250, 500, 750, 1000)

plot_signal_realizations <- function() {
  n <- 1
  for (c in cs) {
    for (s in ss) {
      ms <- samples[['mu_signal']][c, s]
      ss <- samples[['sigma_signal']][c, s]
      l <- samples[['lambda']][c, s]
```

```
      ys <- (1 - l) * dcauchy(xs, ms, ss)
      lines(xs, ys, lwd=2, col=line_colors[n])
      n <- n + 1
    }
  }
  ys <- (1 - lambda) * dcauchy(xs, mu_signal, sigma_signal)
  lines(xs, ys, lwd=4, col="white")
  lines(xs, ys, lwd=2, col=util$c_dark_teal)
}

plot_back_realizations <- function() {
  n <- 1
  for (c in cs) {
    for (s in ss) {
      bb <- samples[['beta_back']][c, s]
      l <- samples[['lambda']][c, s]

      ys <- l * dexp(xs, 1 / bb)
      lines(xs, ys, lwd=2, col=line_colors[n])
      n <- n + 1
    }
  }
  ys <- lambda * dexp(xs, 1 / beta_back)
  lines(xs, ys, lwd=4, col="white")
  lines(xs, ys, lwd=2, col=util$c_dark_teal)
}

plot_sum_realizations <- function() {
  n <- 1
  for (c in cs) {
    for (s in ss) {
      ms <- samples[['mu_signal']][c, s]
      ss <- samples[['sigma_signal']][c, s]
      bb <- samples[['beta_back']][c, s]
      l <- samples[['lambda']][c, s]

      ys <- l * dexp(xs, 1 / bb) + (1 - l) * dcauchy(xs, ms, ss)
      lines(xs, ys, lwd=2, col=line_colors[n])
      n <- n + 1
    }
  }
  ys <- (1 - lambda) * dcauchy(xs, mu_signal, sigma_signal) +
```

```
        lambda * dexp(xs, 1 / beta_back)
  lines(xs, ys, lwd=4, col="white")
  lines(xs, ys, lwd=2, col=util$c_dark_teal)
}
```

While the posterior uncertainties are large we do seem to be able to resolve the underlying signal through the overwhelming background!

```
par(mfrow=c(1, 3), mar=c(5, 5, 2, 1))

xs <- seq(0, 100, 0.5)

plot(NULL, main="Weighted Signal",
     xlab="y", ylab="Probability Density",
     xlim=range(xs), ylim=c(0, 0.05))
plot_signal_realizations()

plot(NULL, main="Weighted Background",
     xlab="y", ylab="Probability Density",
     xlim=range(xs), ylim=c(0, 0.05))
plot_back_realizations()

plot(NULL, main="Mixture",
     xlab="y", ylab="Probability Density",
     xlim=range(xs), ylim=c(0, 0.05))
plot_sum_realizations()
```



In many scientific applications the relative proportion of signal and background, and hence the component probabilities, are more relevant than the probability that any particular observation arose from either source. Fortunately when the latter is of interest computing posterior assignment probabilities and simulating assignments is straightforward.

```
fit <- stan(file="stan_programs/signal_background2.stan",
            data=data, seed=8438338,
            warmup=1000, iter=2024, refresh=0)
```

Because the modified `generated quantities` block consumes pseudo-random numbers differently we need to double check the computational diagnostics. We do see a step size adaptation warning which suggests that the adaptation has changed in one of the Markov chains, but on its own this isn't too problematic.

```
diagnostics <- util$extract_hmc_diagnostics(fit)
util$check_all_hmc_diagnostics(diagnostics)
```

```
  Chain 1: Average proxy acceptance statistic (0.676)
           is smaller than 90% of the target (0.801).

  A small average proxy acceptance statistic indicates that the
```

```
adaptation of the numerical integrator step size failed to converge.
This is often due to discontinuous or imprecise gradients.
```

```
samples <- util$extract_expectand_vals(fit)
base_samples <- util$filter_expectands(samples,
                                       c('mu_signal', 'sigma_signal',
                                         'beta_back', 'lambda'))
util$check_all_expectand_diagnostics(base_samples)
```

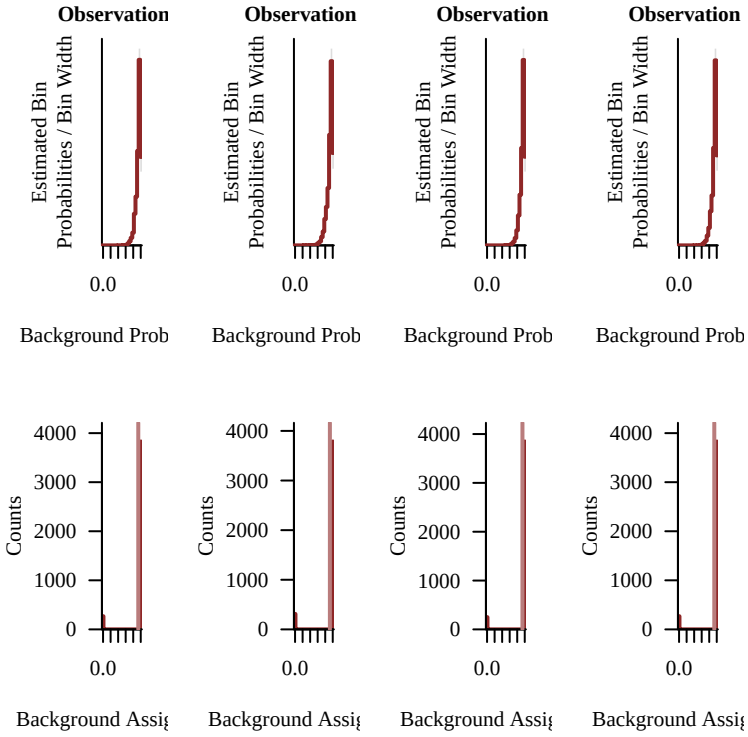
```
mu_signal:
  Chain 1: Right tail hat{xi} (0.516) exceeds 0.25.
  Chain 3: Right tail hat{xi} (0.420) exceeds 0.25.
  Chain 4: Both left and right tail hat{xi}s (0.252, 0.471) exceed 0.25.


Large tail hat{xi}s suggest that the expectand might not be
sufficiently integrable.
```

Consequently we can move on to analyzing the new behaviors.

```
plot_assignment <- function(idxs) {
  for (idx in idxs) {
    name <- paste0('p[', idx, ']')
    util$plot_expectand_pushforward(samples[[name]], 25,
                                    flim=c(-0.03, 1.03),
                                    display_name='Background Probability',
                                    main=paste("Observation", idx))
  }

  for (idx in idxs) {
    name <- paste0('z_pred[', idx, ']')
    zs <- c(samples[[name]], recursive=TRUE)
    util$plot_line_hist(zs, -0.03, 1.03, 0.02,
                        col=util$c_dark,
                        xlab="Background Assignment")
    abline(v=mean(zs), lwd=2, col=util$c_mid)
  }
}
```

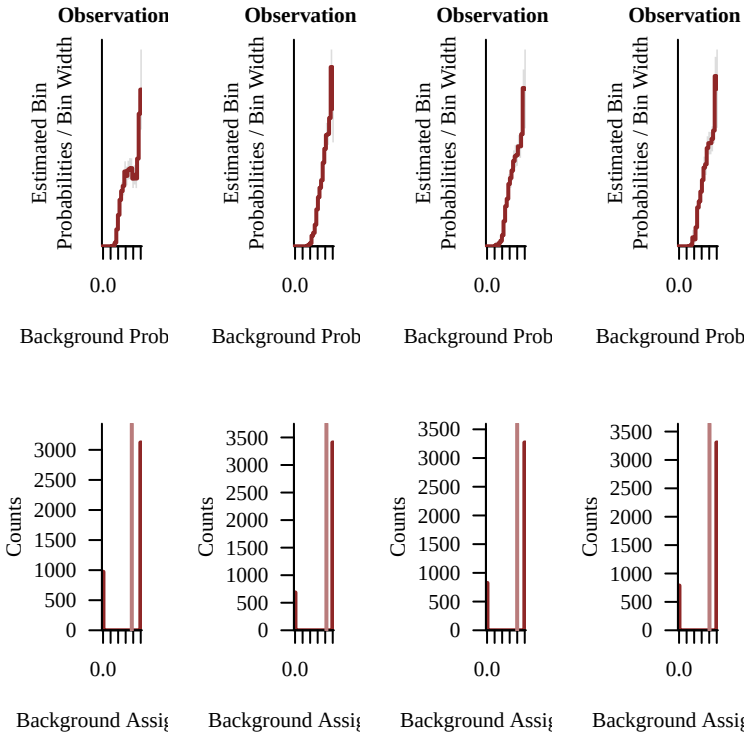
Away from the peak of the signal the observations are strongly associated with the background model, with large individual background probabilities and a predominance of $z = 1$ samples.

```
par(mfrow=c(2, 4), mar=c(5, 5, 2, 1))

idxs <- which(60 < data$y)
plot_assignment(idxs[1:4])
```



Right at its peak the signal model has more of an influence, although the background model still dominates given its much higher base rate.

```
par(mfrow=c(2, 4), mar=c(5, 5, 2, 1))

idxs <-  which(45 < data$y & data$y < 55)
plot_assignment(idxs[1:4])
```



In general the component probabilities are always more informative than the sampled assignments. Consequently they are usually the best way to quantify the behavior of individual observations.

5.3 Zero-Inflated Poisson Model

To demonstrate discrete inflation models let's next consider a zero-inflated Poisson observational model, often affectionately referred to as a "ZIP". Once again we begin by simulating data from a particular configuration of the model.

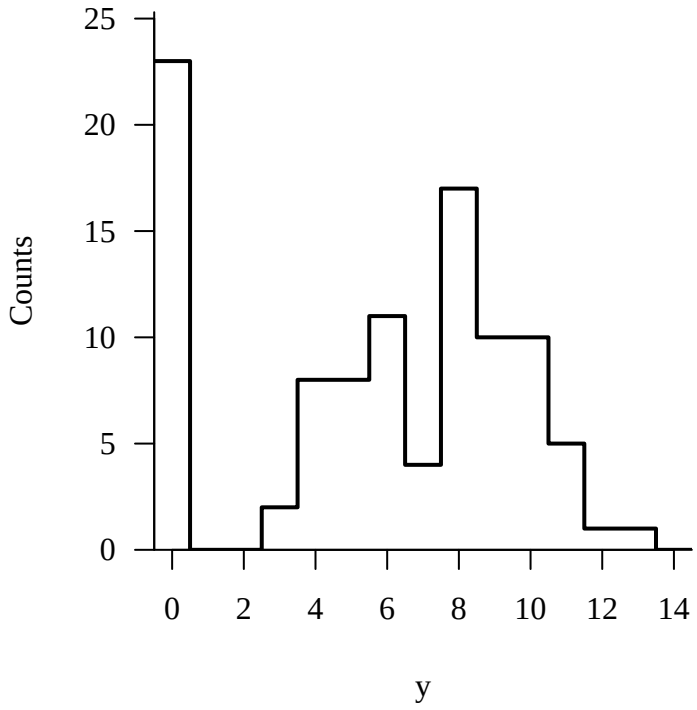
Let's start by simulating data from a configuration where the two component models are well-separated.

```
N <- 100
mu_true <- 7.5
lambda_true <- 0.8

simu <- stan(file="stan_programs/simu_zip.stan",
             algorithm="Fixed_param",
             data=list("N" = N,
                       "mu" = mu_true,
```

```
                          "lambda" = lambda_true),
              seed=8438338,
              warmup=0, iter=1, chains=1, refresh=0)

data <- list("N" = N,
             "y" = extract(simu)$y[1,])
```

Unsurprisingly we see two clear peaks in the observed data, one concentrating entirely at zero and one scattered across larger values.

```
par(mfrow=c(1, 1), mar=c(5, 5, 1, 1))

util$plot_line_hist(data$y,
                    bin_min=-0.5, bin_max=14.5, bin_delta=1,
                    xlab="y")
```



Because the data so clearly separate into two peaks we might hope that inferences will be straightforward.

```
fit <- stan(file="stan_programs/zip1.stan",
            data=data, seed=8438338,
            warmup=1000, iter=2024, refresh=0)
```

Our first good sign is that there are no diagnostics warnings.

```
diagnostics <- util$extract_hmc_diagnostics(fit)
util$check_all_hmc_diagnostics(diagnostics)
```

```
  All Hamiltonian Monte Carlo diagnostics are consistent with reliable
Markov chain Monte Carlo.
```

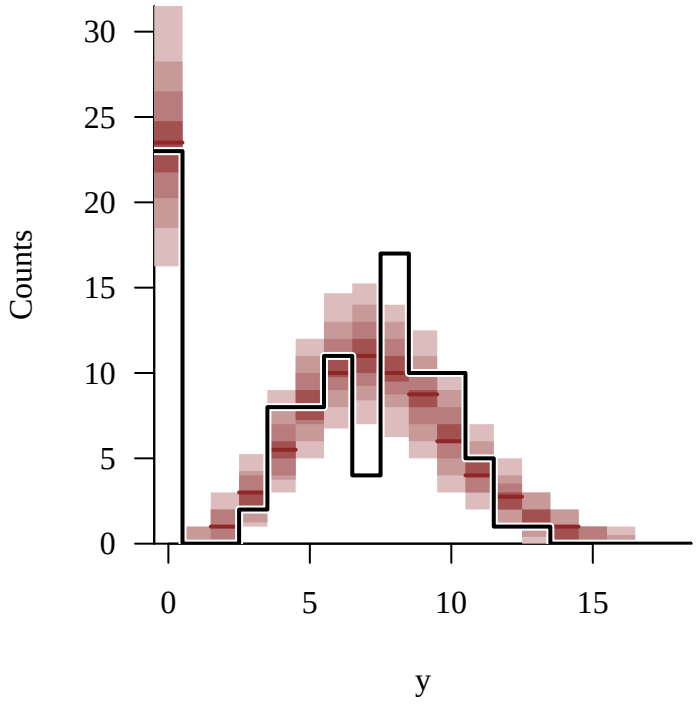
```
samples <- util$extract_expectand_vals(fit)
base_samples <- util$filter_expectands(samples, c('mu', 'lambda'))
util$check_all_expectand_diagnostics(base_samples)
```

```
All expectands checked appear to be behaving well enough for reliable
Markov chain Monte Carlo estimation.
```

A retrodictive check with a histogram summary statistic also looks good, although retrodictive checks will always be pretty well-behaved when we're fitting data simulated from the same model!

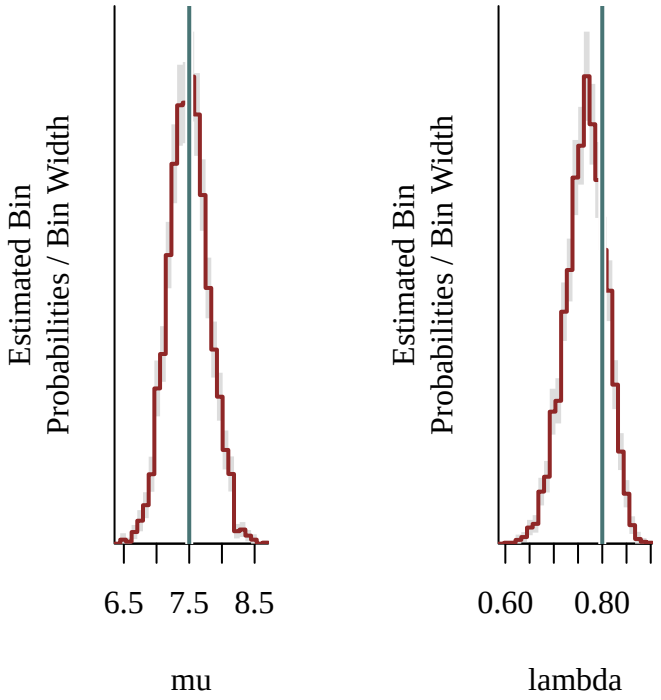
```
par(mfrow=c(1, 1), mar=c(5, 5, 1, 1))

util$plot_hist_quantiles(samples, 'y_pred',
                         bin_min=-0.5, bin_max=18.5, bin_delta=1,
                         baseline_values=data$y, xlab="y")
```

```
Warning in check_bin_containment(bin_min, bin_max, collapsed_values,
"predictive value"): 99 predictive values (0.0%) fell above the binning.
```



More importantly the posterior inferences are able to identify the true model configuration pretty precisely.

```
par(mfrow=c(1, 2), mar=c(5, 5, 1, 1))

util$plot_expectand_pushforward(samples[['mu']], 25,
                                display_name="mu",
                                baseline=mu_true,
                                baseline_col=util$c_mid_teal)

util$plot_expectand_pushforward(samples[['lambda']], 25,
                                display_name="lambda",
                                baseline=lambda_true,
                                baseline_col=util$c_mid_teal)
```



To make things harder let's try again but with much stronger zero inflation.

```
N <- 100
mu_true <- 7.5
lambda_true <- 0.01

simu <- stan(file="stan_programs/simu_zip.stan",
             algorithm="Fixed_param",
             data=list("N" = N,
                       "mu" = mu_true,
                       "lambda" = lambda_true),
             seed=8438338,
             warmup=0, iter=1, chains=1, refresh=0)
```

Indeed the inflation is so strong that the simulated data is comprised entirely of zeros.

```
data <- list("N" = N,
             "y" = extract(simu)$y[1,])

table(data$y)
```

```
  0
100
```

What can we learn about the Poisson component model in this case?

```
fit <- stan(file="stan_programs/zip1.stan",
            data=data, seed=8438338,
            warmup=1000, iter=2024, refresh=0)
```

Unfortunately the diagnostics indicate some weak computational problems.

```
diagnostics <- util$extract_hmc_diagnostics(fit)
util$check_all_hmc_diagnostics(diagnostics)
```

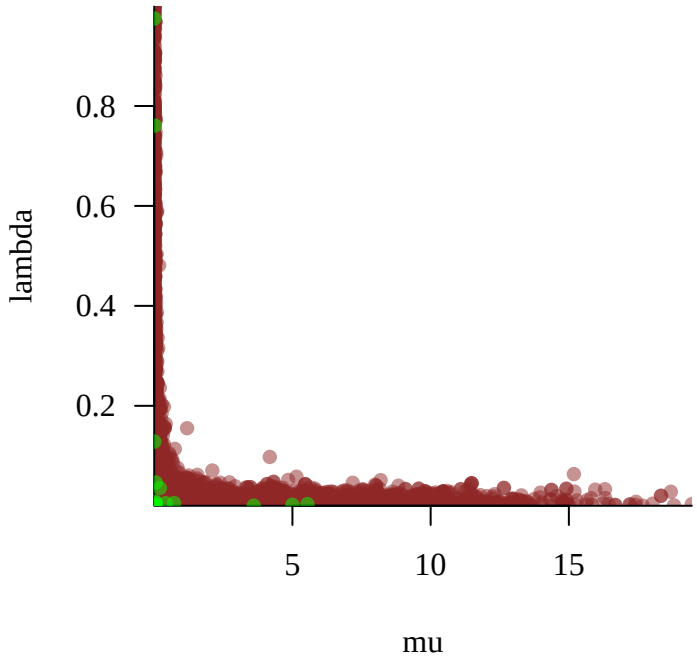
```
  Chain 1: 8 of 1024 transitions (0.8%) diverged.

  Chain 2: 4 of 1024 transitions (0.4%) diverged.

  Chain 4: 2 of 1024 transitions (0.2%) diverged.

  Divergent Hamiltonian transitions result from unstable numerical
trajectories.  These instabilities are often due to degenerate target
geometry, especially "pinches".  If there are only a small number of
divergences then running with adept_delta larger than 0.801 may reduce
the instabilities at the cost of more expensive Hamiltonian
transitions.
```

```
samples <- util$extract_expectand_vals(fit)
base_samples <- util$filter_expectands(samples, c('mu', 'lambda'))
util$check_all_expectand_diagnostics(base_samples)
```

```
lambda:
  Chain 1: Right tail hat{xi} (0.899) exceeds 0.25.
  Chain 2: Right tail hat{xi} (0.892) exceeds 0.25.
  Chain 3: Right tail hat{xi} (1.036) exceeds 0.25.
  Chain 4: Right tail hat{xi} (1.222) exceeds 0.25.


Large tail hat{xi}s suggest that the expectand might not be
sufficiently integrable.
```

Following best practices we'll follow up on the divergences by examining some relevant pair plots. Fortunately here there are only two parameters and hence one possible pair plot to consider.

```
util$plot_div_pairs(x_names="mu", y_names="lambda",
                    samples, diagnostics)
```



One immediate take away from this plot is the extreme posterior uncertainties. The zero-inflated Poisson observational model can accommodate zeros in two distinct ways. Firstly it can push λ to zero, turning off the baseline Poisson component but also leaving the intensity parameter μ uninformed beyond the prior model. Secondly it can push μ to zero so that the two component models become redundant, in which case λ becomes uninformed beyond the prior model. Ultimately the zero-inflated Poisson model with our initial, diffuse prior model is just too flexible to yield well-behaved inferences.

One way to avoid these strong uncertainties, and the resulting strain on the posterior computation, is to constrain the mixture observational model with additional domain expertise. For example any information on the strength of the inflation can inform a more concentrated prior model for λ. Similarly any information on the precise value of μ may be able to suppress configurations where the two component models overlap.

Here let's assume that our domain expertise is inconsistent with values of μ below one. The only problem is that we now need a prior model for μ that suppresses values both above 15 and below 1. Multiple families of probability density functions are applicable here, including the log normal, gamma, and inverse gamma families.

All of these families feature slightly different tail behaviors that have different advantages and disadvantages. For this analysis we'll go with the inverse gamma family as it more heavily suppresses the smaller values of μ where we know the component models become redundant.

To inform a particular inverse gamma configuration we can use Stan's algebraic solver to find the configuration matching our desired tail behaviors.

```
stan(file='stan_programs/prior_tune.stan',
     data=list("y_low" = 1, "y_high" = 15),
     iter=1, warmup=0, chains=1,
     seed=4838282, algorithm="Fixed_param")
```

```
alpha = 3.48681
beta = 9.21604

SAMPLING FOR MODEL 'anon_model' NOW (CHAIN 1).
Chain 1: Iteration: 1 / 1 [100%]  (Sampling)
Chain 1:
Chain 1:  Elapsed Time: 0 seconds (Warm-up)
Chain 1:                0 seconds (Sampling)
Chain 1:                0 seconds (Total)
Chain 1:

Inference for Stan model: anon_model.
1 chains, each with iter=1; warmup=0; thin=1;
post-warmup draws per chain=1, total post-warmup draws=1.

      mean se_mean sd 2.5%  25%  50%  75% 97.5% n_eff Rhat
alpha 3.49      NA NA 3.49 3.49 3.49 3.49  3.49     0  NaN
beta  9.22      NA NA 9.22 9.22 9.22 9.22  9.22     0  NaN
lp__  0.00      NA NA 0.00 0.00 0.00 0.00  0.00     0  NaN

Samples were drawn using (diag_e) at Wed Oct  9 23:02:06 2024.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

With a more informative prior model in hand let's try again.

```
fit <- stan(file="stan_programs/zip2.stan",
            data=data, seed=8438338,
            warmup=1000, iter=2024, refresh=0)
```

There is a lone $\hat{\xi}$ warning, but more importantly the divergences are gone.

```
diagnostics <- util$extract_hmc_diagnostics(fit)
util$check_all_hmc_diagnostics(diagnostics)
```

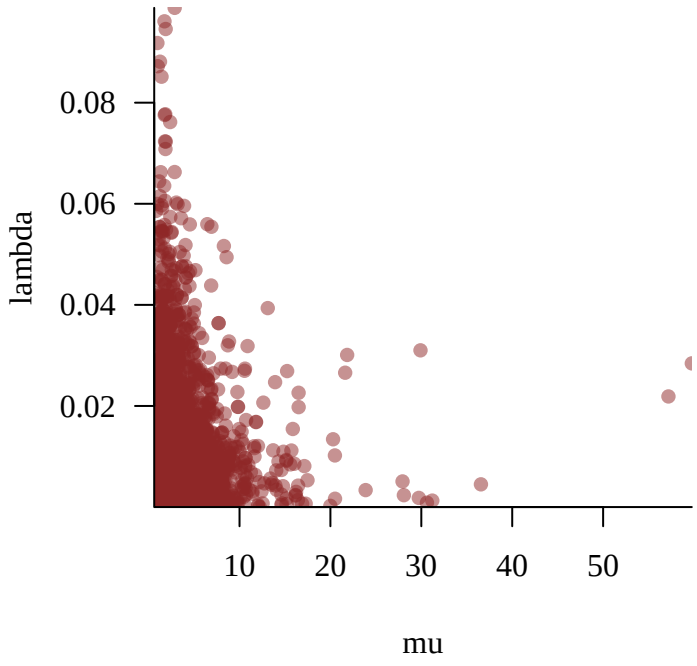
```
  All Hamiltonian Monte Carlo diagnostics are consistent with reliable
Markov chain Monte Carlo.
```

```
samples <- util$extract_expectand_vals(fit)
base_samples <- util$filter_expectands(samples, c('mu', 'lambda'))
util$check_all_expectand_diagnostics(base_samples)
```

```
mu:
  Chain 1: Right tail hat{xi} (0.310) exceeds 0.25.


Large tail hat{xi}s suggest that the expectand might not be
sufficiently integrable.
```

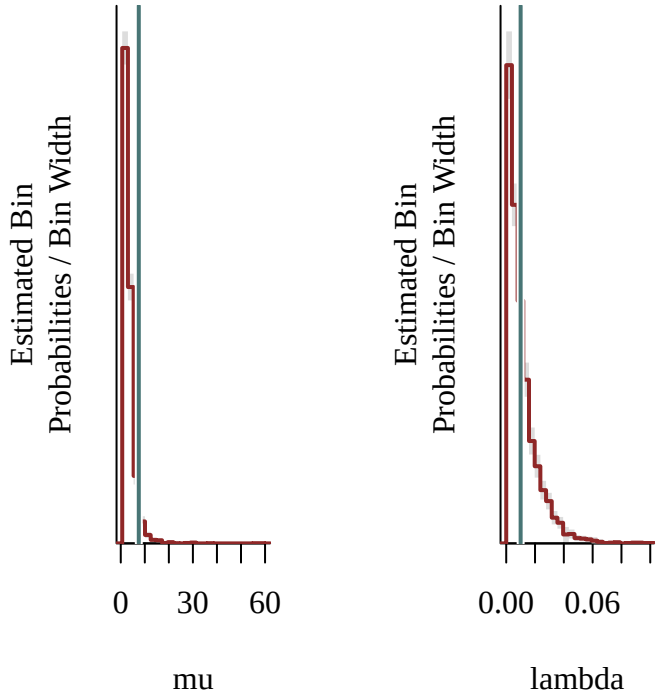
Taking a quick look at the one relevant pair plot we see that the stronger prior model entirely suppresses the ridge where μ is small and λ is poorly informed.

```
util$plot_div_pairs(x_names="mu", y_names="lambda",
                    samples, diagnostics)
```



That said while the posterior geometry is more well-behaved the inferences still leave much to desired. In particular with this more informative prior model the observations no longer inform μ.

```
par(mfrow=c(1, 2), mar=c(5, 5, 1, 1))

util$plot_expectand_pushforward(samples[['mu']], 25,
                                display_name="mu",
                                baseline=mu_true,
                                baseline_col=util$c_mid_teal)

util$plot_expectand_pushforward(samples[['lambda']], 25,
                                display_name="lambda",
                                baseline=lambda_true,
                                baseline_col=util$c_mid_teal)
```



5.4 Zero/One-Inflated Beta Model

Now let's explore what happens when we try to inflate a continuous baseline observational model. Continuous inflation models are often useful for modeling contamination in a data generating process, such as data-entry errors or unexpected/corrupted outcomes that have been coded with default values. For example data entry software that fills in all entries with

zeros before allowing a user to overwrite that default value will give excess zeros if the user fails to enter all observed values. Similar missing observations that are coded with zero will also result in an excess of zeros.

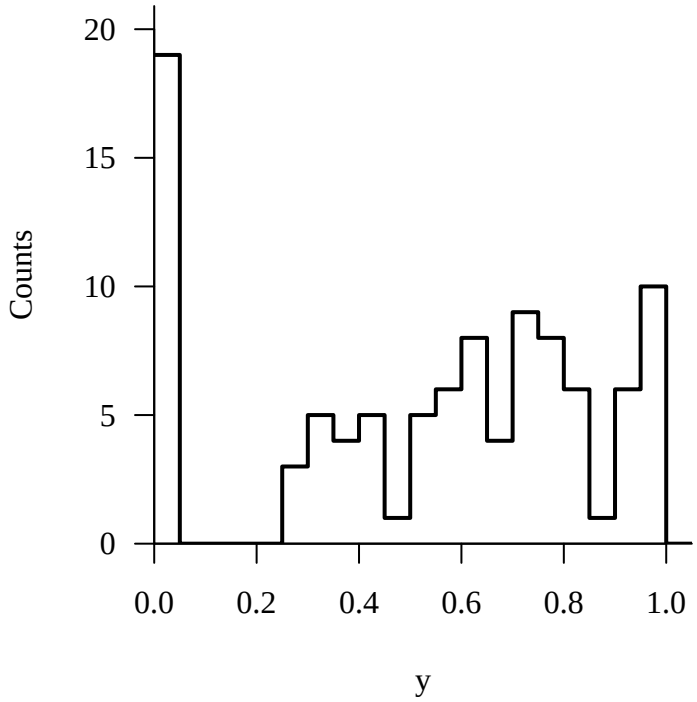
Here let's inflate both zero and one values in baseline beta observational model, giving what is known as a zero/one-inflated beta model. While less conventional than "ZIP" for a zero-inflated Poisson model, the short-hand "ZOIB" for this model is advocated by a small but passionate group.

Simulating data from a continuous inflation observational model proceeds exactly the same as for a discrete mixture model, and indeed any mixture model.

```
N <- 100

simu <- stan(file="stan_programs/simu_zoib.stan",
             algorithm="Fixed_param",
             data=list("N" = N), seed=8438338,
             warmup=0, iter=1, chains=1, refresh=0)

data <- list("N" = N,
             "y" = extract(simu)$y[1,])
```

Because of the finite binning histogram visualizations of the data can make it difficult to distinguish between inflation *near* zero and one and inflation *exactly at* zero and one.

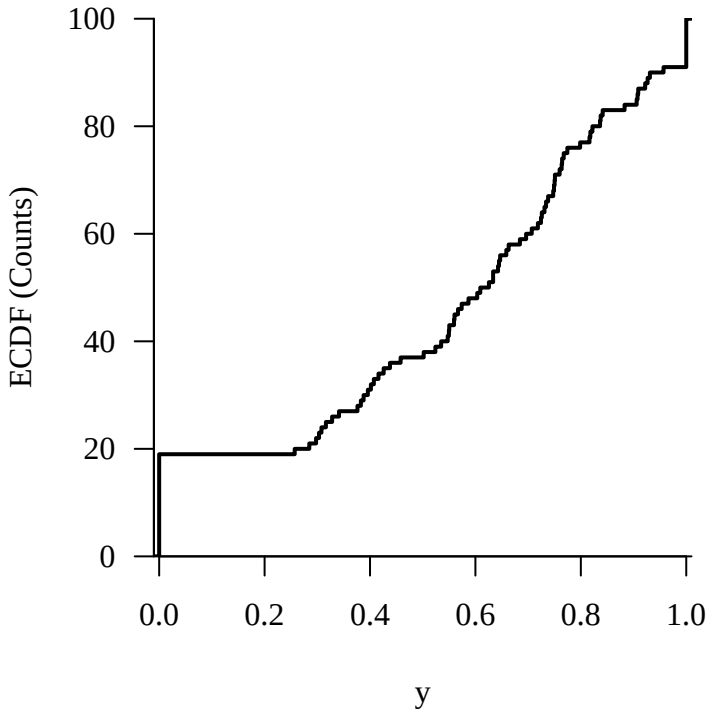
```
par(mfrow=c(1, 1), mar=c(5, 5, 1, 1))

util$plot_line_hist(data$y,
                    bin_min=0, bin_max=1.05, bin_delta=0.05,
                    xlab="y")
```



Empirical cumulative distribution functions, are better suited to identifying continuous inflation, which manifests as distinct jumps in the cumulative probabilities.

```
plot_ecdf <- function(vals, delta=0.01, xlab="") {
  N <- length(vals)
  ordered_vals <- sort(vals)

  xs <- c(ordered_vals[1] - delta,
          rep(ordered_vals, each=2),
          ordered_vals[N] + delta)

  ecdf_counts <- rep(0:N, each=2)

  plot(xs, ecdf_counts, type="l", lwd="2", col="black",
       xlab=xlab,
       ylim=c(0, N), ylab="ECDF (Counts)")
}
```

```
plot_ecdf(data$y, xlab="y")
```



As discussed in Section 2.1.2 the implementation of a continuous inflation model requires treating the inflation values as a discrete space separate from the other values. There are two ways to handle this.

Firstly we can model the inflated and non-inflated values jointly.

```
fit <- stan(file="stan_programs/zoib1.stan",
            data=data, seed=8438338,
            warmup=1000, iter=2024, refresh=0)
```

The diagnostics show no problems.

```
diagnostics <- util$extract_hmc_diagnostics(fit)
util$check_all_hmc_diagnostics(diagnostics)
```

```
  All Hamiltonian Monte Carlo diagnostics are consistent with reliable
Markov chain Monte Carlo.
```

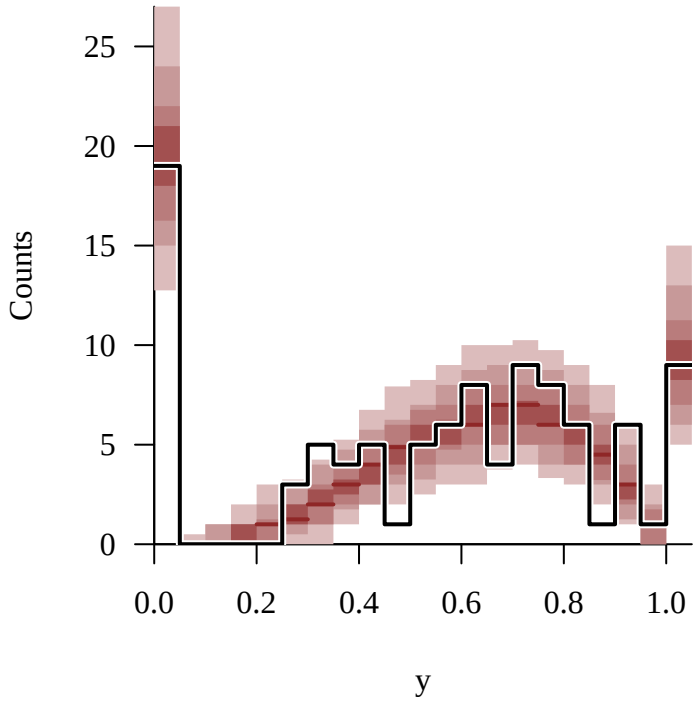
```
samples1 <- util$extract_expectand_vals(fit)
base_samples <- util$filter_expectands(samples1,
                                       c('alpha', 'beta', 'lambda'),
```

```
                                          check_arrays=TRUE)
util$check_all_expectand_diagnostics(base_samples)
```

```
All expectands checked appear to be behaving well enough for reliable
Markov chain Monte Carlo estimation.
```

Nor does the posterior retrodictive check.

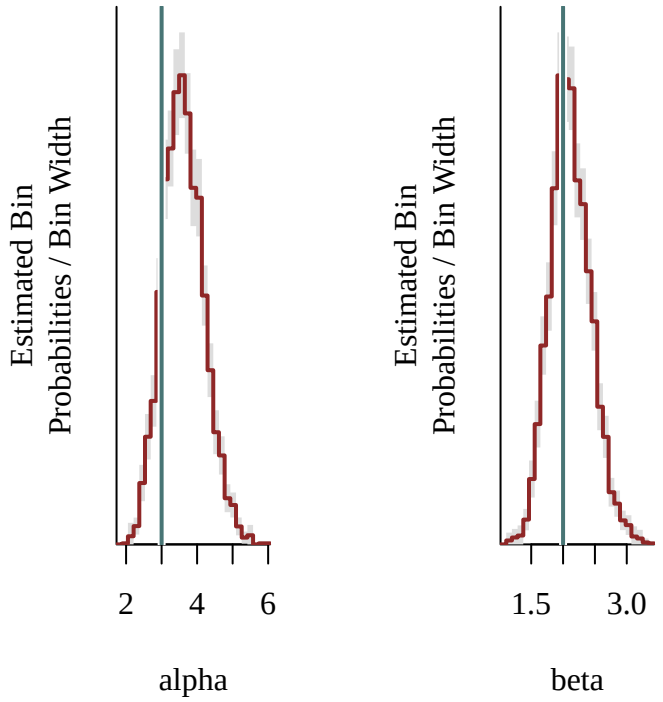
```
par(mfrow=c(1, 1), mar=c(5, 5, 1, 1))

util$plot_hist_quantiles(samples1, 'y_pred',
                         bin_min=0, bin_max=1.05, bin_delta=0.05,
                         baseline_values=data$y, xlab="y")
```



Moreover our posterior inferences are both precise and accurate to the simulation data generating process. For instance the inferences are consistent with the true configuration of the baseline beta model used to simulate the data.

```
par(mfrow=c(1, 2), mar=c(5, 5, 1, 1))

util$plot_expectand_pushforward(samples1[['alpha']], 25,
```

```
                                    display_name="alpha",
                                    baseline=3,
                                    baseline_col=util$c_mid_teal)

util$plot_expectand_pushforward(samples1[['beta']], 25,
                                    display_name="beta",
                                    baseline=2,
                                    baseline_col=util$c_mid_teal)
```



Because we have three component models we can directly visualize the simplex of component probabilities. This offers a much more complete picture of our inferences than trying to visualize the marginal inferences for each component probability independently and neglecting the simplex constraints.

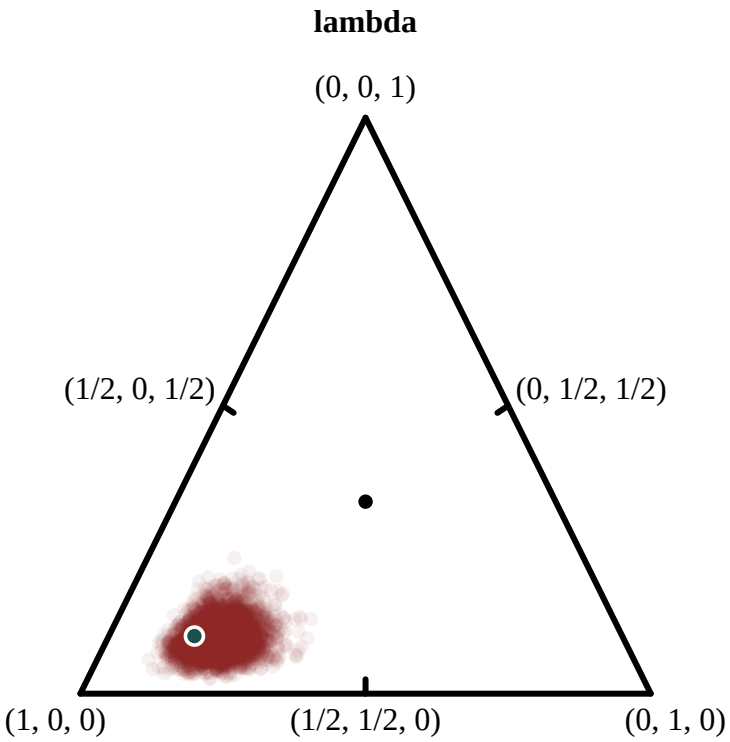
```
to_plot_coordinates <- function(q, C) {
  c(C * (q[2] - q[1]), q[3])
}

plot_simplex_border <- function(label_cex, C, center_label=TRUE) {
  lines( c(-C, 0), c(0, 1), lwd=3)
  lines( c(+C, 0), c(0, 1), lwd=3)
  lines( c(-C, +C), c(0, 0), lwd=3)
```

```
  text_delta <- 0.05
  text( 0, 1 + text_delta, "(0, 0, 1)", cex=label_cex)
  text(-C - text_delta, -text_delta, "(1, 0, 0)", cex=label_cex)
  text(+C + text_delta, -text_delta, "(0, 1, 0)", cex=label_cex)

  tick_delta <- 0.025
  lines( c(0, 0), c(0, tick_delta), lwd=3)
  text(0, 0 - text_delta, "(1/2, 1/2, 0)", cex=label_cex)

  lines( c(+C * 0.5, +C * 0.5 - tick_delta * 0.5 * sqrt(3)),
         c(0.5, 0.5 - tick_delta * 0.5), lwd=3)
  text(C * 0.5 + text_delta * 0.5 * sqrt(3) + 2.5 * text_delta,
       0.5 + text_delta * 0.5, "(0, 1/2, 1/2)", cex=label_cex)

  lines( c(-C * 0.5, -C * 0.5 + tick_delta * 0.5 * sqrt(3)),
         c(0.5, 0.5 - tick_delta * 0.5), lwd=3)
  text(-C * 0.5 - text_delta * 0.5 * sqrt(3) - 2.5 * text_delta,
       0.5 + text_delta * 0.5, "(1/2, 0, 1/2)", cex=label_cex)

  points(0, 1/3, col="white", pch=16, cex=1.5)
  points(0, 1/3, col="black", pch=16, cex=1)
  if (center_label)
    text(0, 1/3 - 1.5 * text_delta, "(1/3, 1/3, 1/3)", cex=label_cex)
}

plot_simplex_samples <- function(q1, q2, q3, label_cex=1,
                                 main="", baseline=NULL) {
  N <- 200
  C <- 1 / sqrt(3)

  plot(NULL, xlab="", ylab="", xaxt="n", yaxt="n", frame.plot=F,
       xlim=c(-(C + 0.2), +(C + 0.2)), ylim=c(-0.1, 1.1))
  plot_simplex_border(label_cex, C, FALSE)
  title(main)

  N <- min(length(q1), length(q2), length(q3))
  for (n in 1:N) {
    xy <- to_plot_coordinates(c(q1[n], q2[n], q3[n]), C)
    points(xy[1], xy[2], col="#8F272710", pch=16, cex=1.0)
  }
  if (!is.null(baseline)) {
    xy <- to_plot_coordinates(baseline, C)
```

```
    points(xy[1], xy[2], col="white", pch=16, cex=1.5)
    points(xy[1], xy[2], col=util$c_dark_teal, pch=16, cex=1)
  }
}
```

With this visualization we can clearly see the posterior inferences for the component probabilities concentrating around the true value.

```
par(mfrow=c(1, 1), mar=c(0, 0, 2, 0))

plot_simplex_samples(c(samples1[['lambda[1]']], recursive=TRUE),
                     c(samples1[['lambda[2]']], recursive=TRUE),
                     c(samples1[['lambda[3]']], recursive=TRUE),
                     main="lambda", baseline=c(0.75, 0.15, 0.10))
```



While the joint model works well we can also fit the inflated and non-inflated observations independently of each other.

```
data$N_zero <- sum(data$y == 0)
data$N_one  <- sum(data$y == 1)
```

```
data$y_else <- data$y[data$y != 0 & data$y != 1]
data$N_else <- length(data$y_else)
```

```
fit <- stan(file="stan_programs/zoib2a.stan",
            data=data, seed=8438338,
            warmup=1000, iter=2024, refresh=0)

diagnostics <- util$extract_hmc_diagnostics(fit)
util$check_all_hmc_diagnostics(diagnostics)
```

```
  All Hamiltonian Monte Carlo diagnostics are consistent with reliable
Markov chain Monte Carlo.
```

```
samples2a <- util$extract_expectand_vals(fit)
util$check_all_expectand_diagnostics(samples2a)
```

```
All expectands checked appear to be behaving well enough for reliable
Markov chain Monte Carlo estimation.
```

```
fit <- stan(file="stan_programs/zoib2b.stan",
            data=data, seed=8438338,
            warmup=1000, iter=2024, refresh=0)

diagnostics <- util$extract_hmc_diagnostics(fit)
util$check_all_hmc_diagnostics(diagnostics)
```

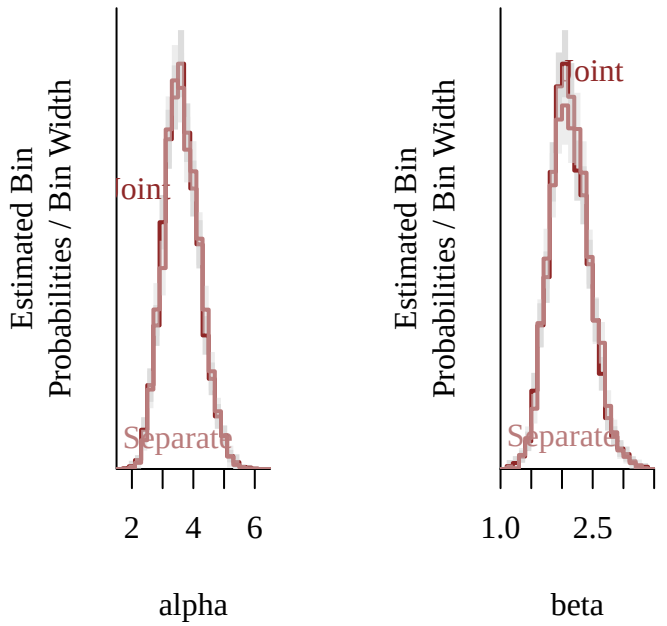
```
  All Hamiltonian Monte Carlo diagnostics are consistent with reliable
Markov chain Monte Carlo.
```

```
samples2b <- util$extract_expectand_vals(fit)
util$check_all_expectand_diagnostics(samples2b)
```

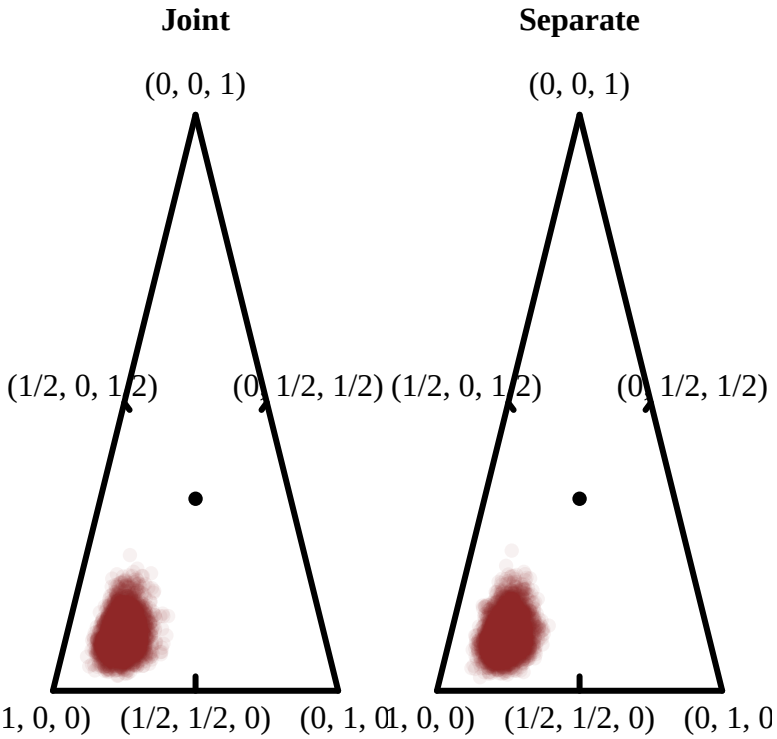
```
All expectands checked appear to be behaving well enough for reliable
Markov chain Monte Carlo estimation.
```

Critically the posterior inferences derived in the two approaches are consistent up to the expected Markov chain Monte Carlo variation.

```
par(mfrow=c(1, 2), mar=c(5, 5, 3, 1))

util$plot_expectand_pushforward(samples1[['alpha']],
                                25, flim=c(1.5, 6.5),
                                display_name="alpha")
text(2.25, 0.5, "Joint", col=util$c_dark)

util$plot_expectand_pushforward(samples2a[['alpha']],
                                25, flim=c(1.5, 6.5),
                                col=util$c_mid,
                                border="#DDDDDD88",
                                add=TRUE)
text(3.5, 0.05, "Separate", col=util$c_mid)

util$plot_expectand_pushforward(samples1[['beta']],
                                25, flim=c(1, 3.5),
                                display_name="beta",)
text(2.5, 1.3, "Joint", col=util$c_dark)

util$plot_expectand_pushforward(samples2a[['beta']],
                                25, flim=c(1, 3.5),
                                col=util$c_mid,
                                border="#DDDDDD88",
                                add=TRUE)
text(2, 0.1, "Separate", col=util$c_mid)
```




```
par(mfrow=c(1, 2), mar=c(0, 0, 2, 0))

plot_simplex_samples(c(samples1[['lambda[1]']], recursive=TRUE),
                     c(samples1[['lambda[2]']], recursive=TRUE),
                     c(samples1[['lambda[3]']], recursive=TRUE),
                     main="Joint")

plot_simplex_samples(c(samples2b[['lambda[1]']], recursive=TRUE),
                     c(samples2b[['lambda[2]']], recursive=TRUE),
                     c(samples2b[['lambda[3]']], recursive=TRUE),
                     main="Separate")
```



If the inflated values are modeling some undesired contamination and we are interested in only the behavior of the continuous baseline observational model then we can always fit the baseline model to the non-inflated values and ignore the inflated values entirely. Similarly if we are interested in only the prevalence of inflated values then we can fit the component probabilities to the counts directly, ignoring the precise value of the continuous observations.

5.5 Redundant Mixture Model

To avoid any ambiguity let me emphasize that I do not recommend using redundant mixture models in applied practice. Without some way of distinguishing the component probability distributions mixture models are inherently prone to degenerate inferences that impede meaningful insights.

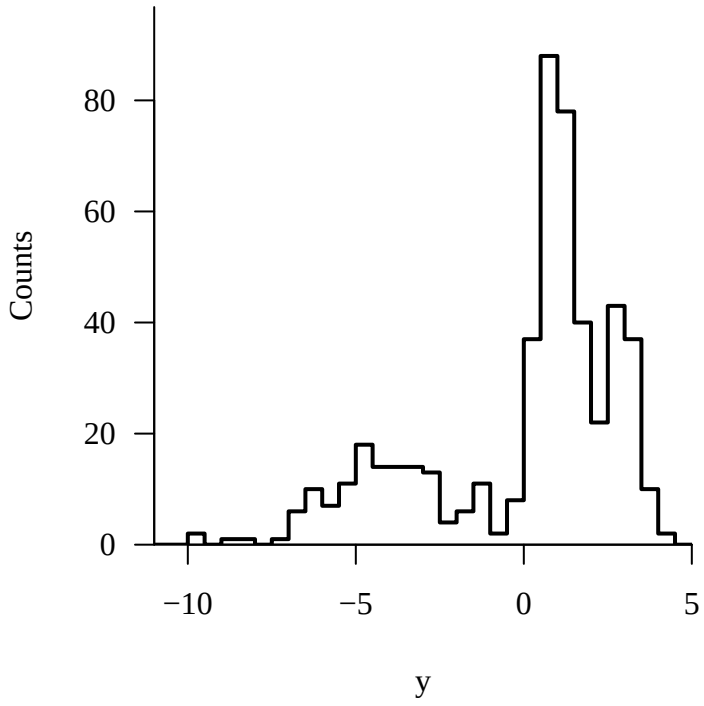
That said, in this section we will explore a redundant mixture of normal observational models to see just how problematic redundancy can be in mixture models. If you're not tempted to use redundant mixture models in your own analyses then please feel free to skim if not skip this section!

At the very least the simulation of data is straightforward.

```
N <- 500

simu <- stan(file="stan_programs/simu_normal_mix.stan",
             algorithm="Fixed_param",
             data=list("N" = N), seed=8438338,
             warmup=0, iter=1, chains=1, refresh=0)

data <- list("N" = N,
             "y" = extract(simu)$y[1,])
```

A histogram of the data clearly shows at least two peaks, with the possibility of the second peak maybe separating into two more peaks.

```
par(mfrow=c(1, 1), mar=c(5, 5, 1, 1))

util$plot_line_hist(data$y, -11, 5, 0.5, xlab="y")
```



5.5.1 Unknown Component Probabilities

If we know the true configuration of the component observational models, so that all we have to infer are the component probabilities, then the components will not actually be redundant.

```
fit <- stan(file="stan_programs/normal_mix1.stan",
            data=data, seed=8438338,
            warmup=1000, iter=2024, refresh=0)
```

In this case the computation is clean.

```
diagnostics <- util$extract_hmc_diagnostics(fit)
util$check_all_hmc_diagnostics(diagnostics)
```

```
  All Hamiltonian Monte Carlo diagnostics are consistent with reliable
Markov chain Monte Carlo.
```

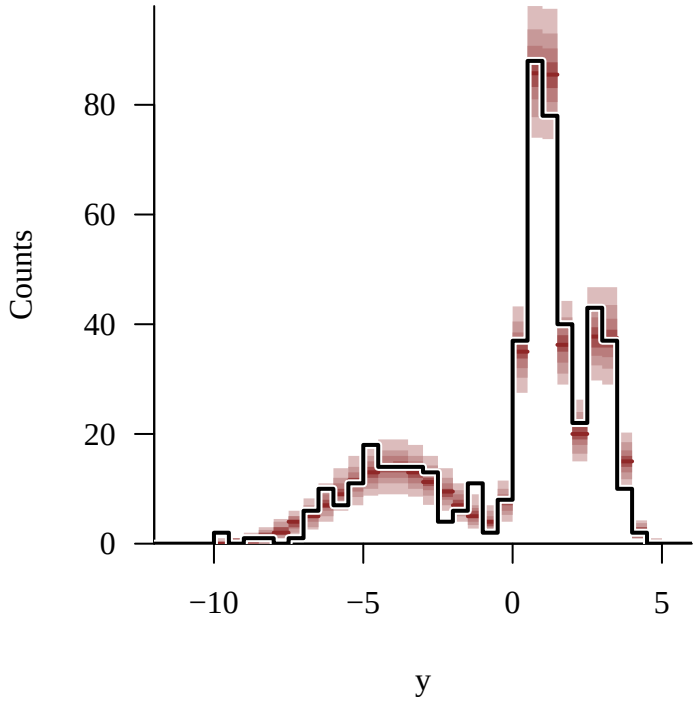
```
samples <- util$extract_expectand_vals(fit)
base_samples <- util$filter_expectands(samples,
                                       c('lambda'),
                                       check_arrays=TRUE)
util$check_all_expectand_diagnostics(base_samples)
```

```
All expectands checked appear to be behaving well enough for reliable
Markov chain Monte Carlo estimation.
```

The posterior retrodictive check shows not only shows signs of model inadequacy but also that our model infers three distinct peaks from the observed data.

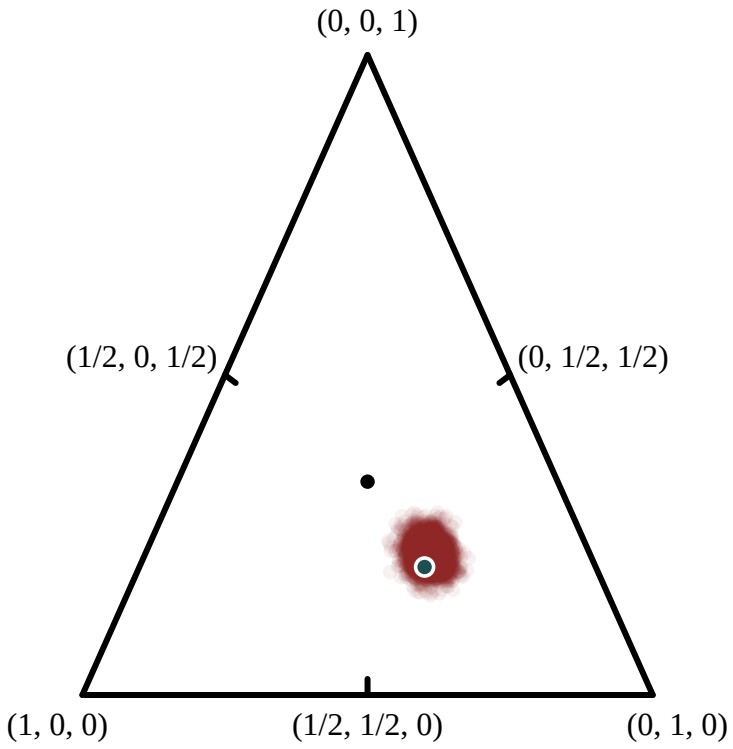
```
par(mfrow=c(1, 1), mar=c(5, 5, 1, 1))

util$plot_hist_quantiles(samples, 'y_pred', -12, 6, 0.5,
                         baseline_values=data$y, xlab="y")
```

```
Warning in check_bin_containment(bin_min, bin_max, collapsed_values,
"predictive value"): 12 predictive values (0.0%) fell below the binning.
```



Moreover posterior inferences for the component probabilities are consistent with the true values used to simulate the data.

```
par(mfrow=c(1, 1), mar=c(0, 0, 0, 0))

lambda_true <- c(0.3, 0.5, 0.2)
plot_simplex_samples(c(samples[['lambda[1]']], recursive=TRUE),
                     c(samples[['lambda[2]']], recursive=TRUE),
                     c(samples[['lambda[3]']], recursive=TRUE),
                     baseline=lambda_true)
```



5.5.2 Unknown Component Probabilities and Locations

Let's now complicate the situation by leaving the component scales fixed but trying to infer the component locations. Because the component scales are not all equal to each other the resulting mixture model is not completely redundant. That said the components are all pretty similar, and the latter two components are exactly redundant with each other.

```
fit <- stan(file="stan_programs/normal_mix2a.stan",
            data=data, seed=8438338,
            warmup=1000, iter=2024, refresh=0)
```

The preponderance of split $\hat{R}$ warnings hints at a multi-modal posterior distribution.

```
diagnostics <- util$extract_hmc_diagnostics(fit)
util$check_all_hmc_diagnostics(diagnostics)
```

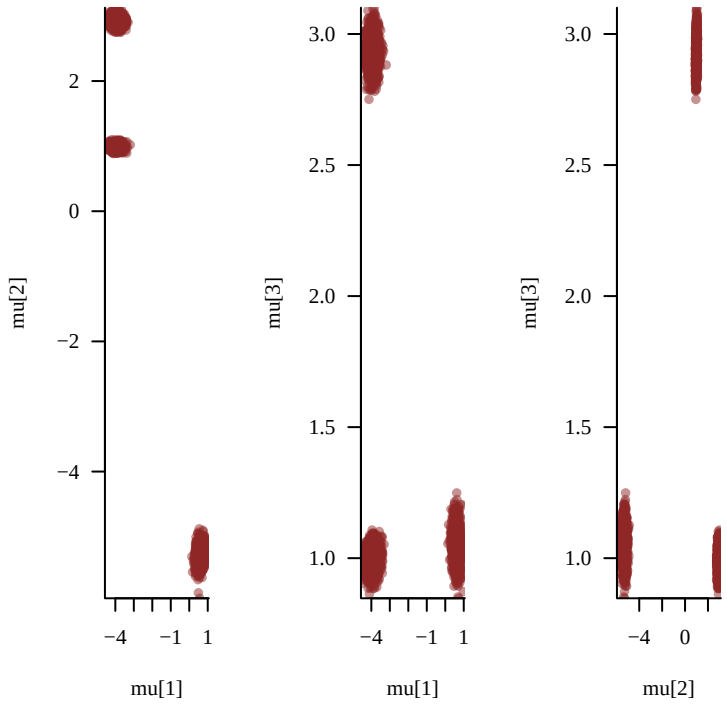
```
  All Hamiltonian Monte Carlo diagnostics are consistent with reliable
Markov chain Monte Carlo.
```

```
samples <- util$extract_expectand_vals(fit)
base_samples <- util$filter_expectands(samples,
                                       c('mu', 'lambda'),
                                       check_arrays=TRUE)
util$check_all_expectand_diagnostics(base_samples)
```

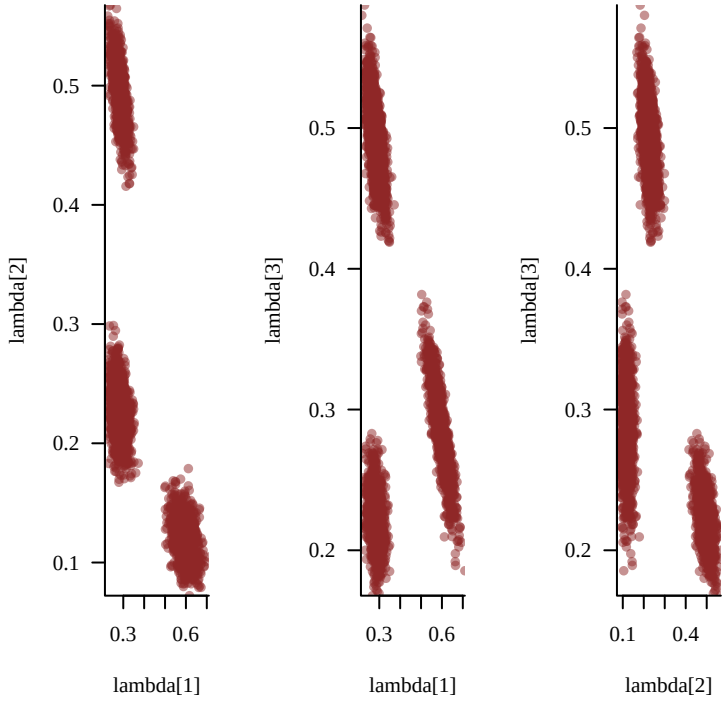
```
mu[1]:
  Split hat{R} (12.002) exceeds 1.1.

mu[2]:
  Split hat{R} (47.876) exceeds 1.1.

mu[3]:
  Split hat{R} (18.290) exceeds 1.1.

lambda[1]:
  Split hat{R} (5.779) exceeds 1.1.

lambda[2]:
  Split hat{R} (7.297) exceeds 1.1.

lambda[3]:
  Split hat{R} (5.198) exceeds 1.1.


Split Rhat larger than 1.1 suggests that at least one of the Markov
chains has not reached an equilibrium.
```

Indeed a selection of pair plots suggest at least three distinct posterior modes, although accurately counting modes based on two-dimensional projections is always tricky. Moreover there could easily be more modes that our Markov chains missed.

```
names <- sapply(1:3, function(k) paste0('mu[', k, ']'))
util$plot_div_pairs(names, names, samples, diagnostics)
```

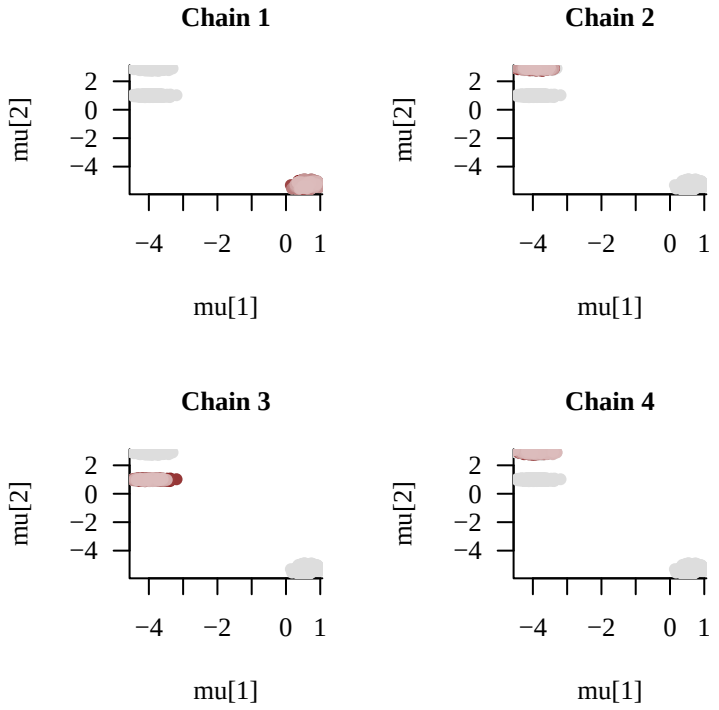



```
names <- sapply(1:3, function(k) paste0('lambda[', k, ']'))
util$plot_div_pairs(names, names, samples, diagnostics)
```



Unsurprisingly the individual Markov chains are each confined to a single mode. Because the first, second, and third Markov chains appear to have fallen into distinct modes we'll focus on those going forwards.

```
util$plot_pairs_by_chain(samples[['mu[1]']], 'mu[1]',
                         samples[['mu[2]']], 'mu[2]')
```



The propensity for individual modes to capture entire Markov chains is both why we cannot accurately estimate the relative importance of each mode and why running many independent Markov chains is the most practical way to diagnose multi-modality.

To understand the behavior within each mode let's explore the inferred component behaviors within each individual Markov chain, and hence within individual modes.

```
plot_component_realizations <- function(k, c) {
  n <- 1
  for (s in 50 * (1:20)) {
    mu_name <- paste0('mu[', k, ']')
    mu <- samples[[mu_name]][c, s]

    sigma <- c(2, 0.5, 0.5)[k]

    lambda_name <- paste0('lambda[', k, ']')
```

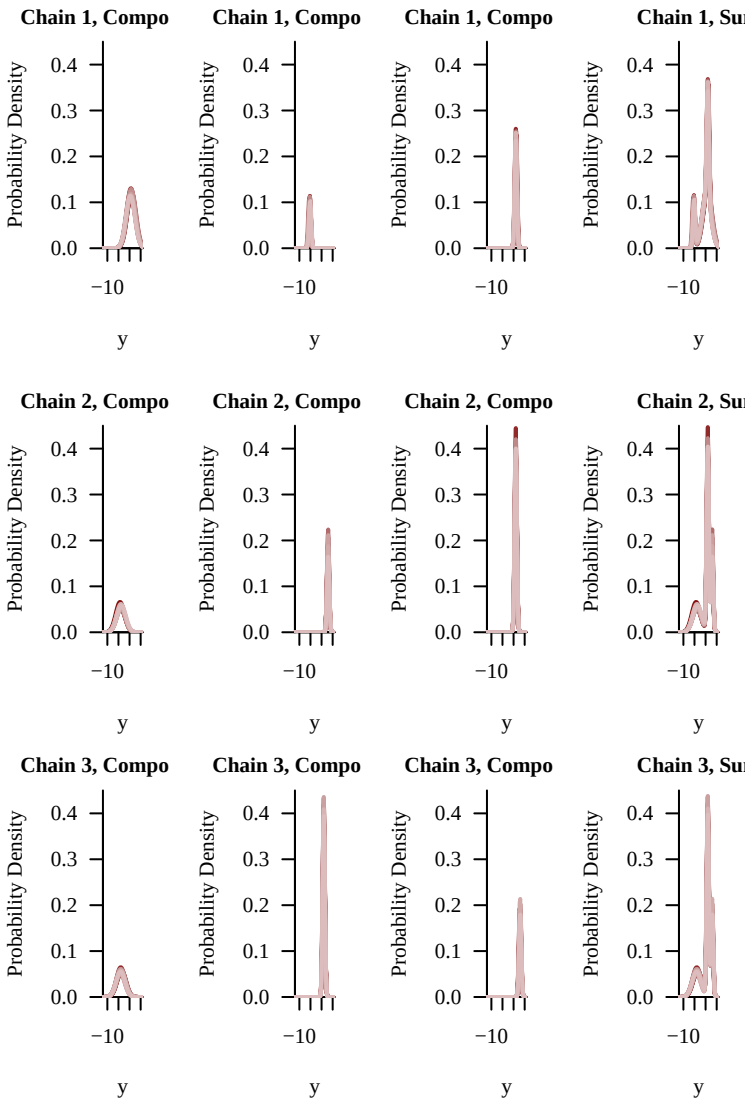
```
    lambda <- samples[[lambda_name]][c, s]

    ys <- lambda * dnorm(xs, mu, sigma)
    lines(xs, ys, lwd=2, col=line_colors[n])
    n <- n + 1
  }
}

plot_sum_realizations <- function(c) {
  n <- 1
  for (s in 50 * (1:20)) {
    mu_names <- sapply(1:3, function(k) paste0('mu[', k, ']'))
    mu <- sapply(mu_names, function(name) samples[[name]][c, s])

    sigma <- c(2, 0.5, 0.5)

    lambda_names <- sapply(1:3, function(k) paste0('lambda[', k, ']'))
    lambda <- sapply(lambda_names, function(name) samples[[name]][c, s])

    ys <- rep(0, length(xs))
    for (k in 1:3) {
      ys <- ys + lambda[k] * dnorm(xs, mu[k], sigma[k])
    }
    lines(xs, ys, lwd=2, col=line_colors[n])
    n <- n + 1
  }
}
```

In the first Markov chain the inferred behaviors of the first two components are swapped relative to the inferences in the second and third Markov chains. Because the component scales are fixed this exchange results in a slightly different mixture density function, but similar enough for both to be somewhat consistent with the observed data.

At the same time the behavior of the second and third components are swapped between the second and third Markov chains. In this case the known scales are the same and the resulting mixture probability density functions are identical.

```
xs <- seq(-12, 6, 0.25)

par(mfrow=c(2, 4), mar=c(5, 5, 2, 1))

for (c in c(1, 2, 3)) {
```

```
  for (k in 1:3) {
    plot(NULL, main=paste0('Chain ', c, ', Component ', k),
         xlab="y", ylab="Probability Density",
         xlim=range(xs), ylim=c(0, 0.45))
    plot_component_realizations(k, c)
  }
  plot(NULL, main=paste0('Chain ', c, ', Sum'),
       xlab="y", ylab="Probability Density",
       xlim=range(xs), ylim=c(0, 0.45))
  plot_sum_realizations(c)
}
```



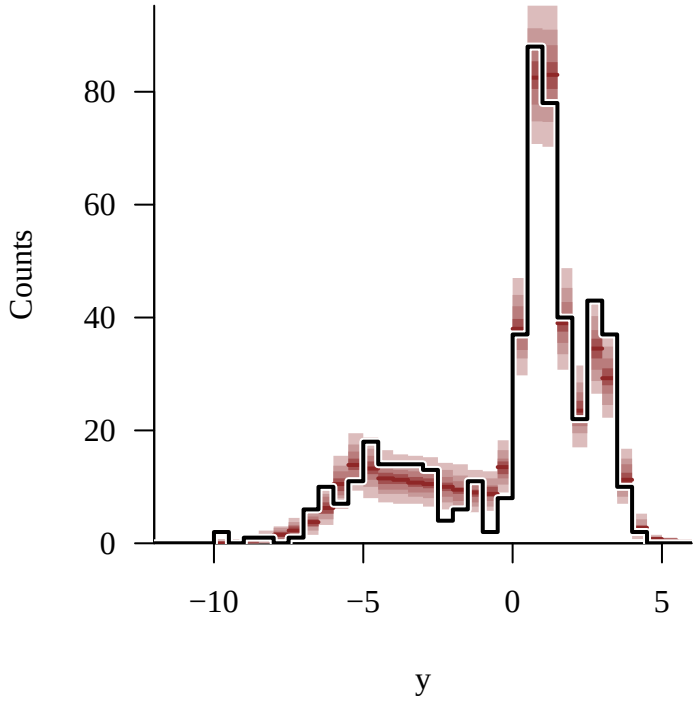
Given the multi-modality the contributions from the individual modes are almost surely not being weighted correctly. Consequently we cannot take our posterior quantification, and the resulting posterior predictive quantification, too seriously. That said, let's see what have.

Overall the retrodictive performance show no signs of problems.

```
par(mfrow=c(1, 1), mar=c(5, 5, 1, 1))

util$plot_hist_quantiles(samples, 'y_pred', -12, 6, 0.5,
                         baseline_values=data$y, xlab="y")
```

```
Warning in check_bin_containment(bin_min, bin_max, collapsed_values,
"predictive value"): 8 predictive values (0.0%) fell below the binning.
```

```
Warning in check_bin_containment(bin_min, bin_max, collapsed_values,
"predictive value"): 1147 predictive values (0.1%) fell above the binning.
```



Because of the multi-modality, however, it's a bit more fair to look at the retrodictive performance within individual Markov chain. Here we see that the retrodictive performance from the first Markov chain is similar to, but not exactly the same as, that from the second and third Markov chains, consistent with the inferred component behaviors. Moreover the retrodictive performance in the second and third Markov chains is the same, consistent with

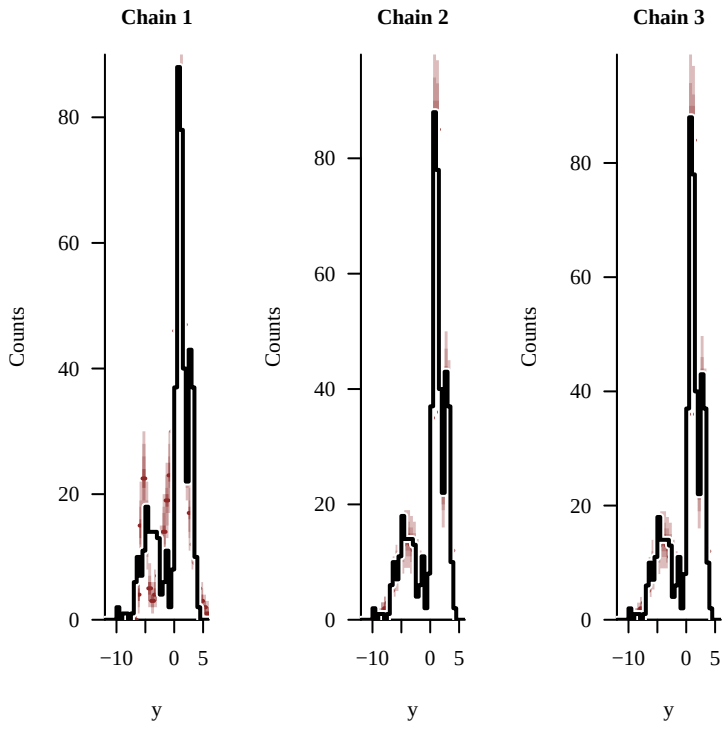
the corresponding modes capturing model configurations that are exact permutations of each other.

```
par(mfrow=c(1, 3), mar=c(5, 5, 3, 1))

for (c in 1:3) {
  ss <- lapply(samples, function(s) array(s[c,], dim=c(1, 1024)))
  util$plot_hist_quantiles(ss, 'y_pred', -12, 6, 0.5,
                           baseline_values=data$y, xlab="y",
                           main=paste0('Chain ', c))
}
```

```
Warning in check_bin_containment(bin_min, bin_max, collapsed_values,
"predictive value"): 1147 predictive values (0.2%) fell above the binning.
```

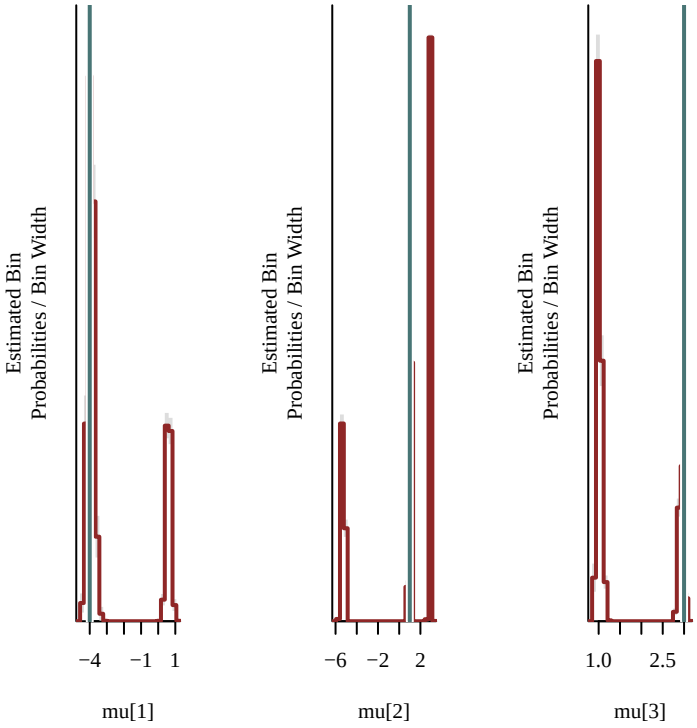
```
Warning in check_bin_containment(bin_min, bin_max, collapsed_values,
"predictive value"): 3 predictive values (0.0%) fell below the binning.
```

```
Warning in check_bin_containment(bin_min, bin_max, collapsed_values,
"predictive value"): 4 predictive values (0.0%) fell below the binning.
```

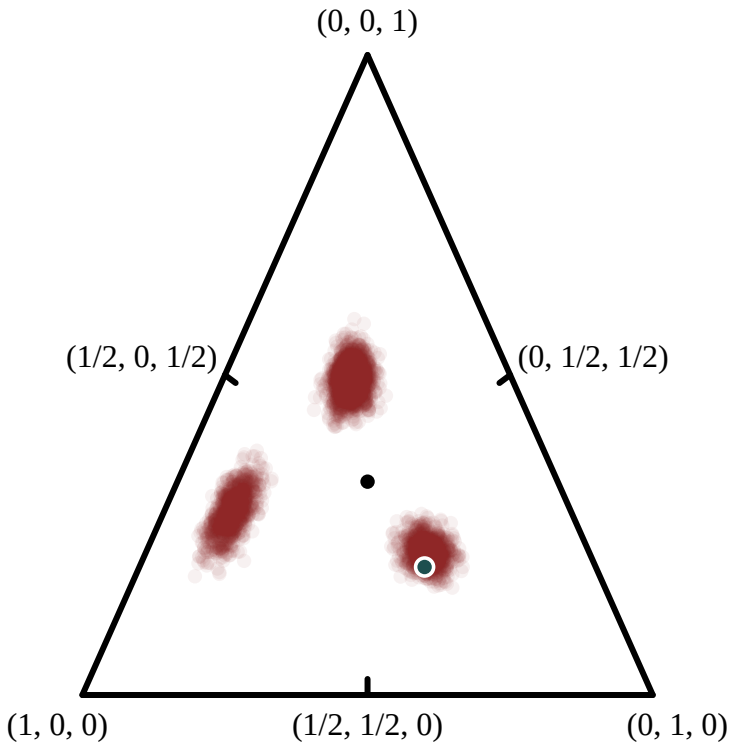


One of the posterior modes does appear to have captured the true model configuration, although again because we cannot rely on the relative weights of those modes we can't be certain how much the exact posterior distribution prefers that one correct mode over the others.

```
par(mfrow=c(1, 3), mar=c(5, 5, 1, 1))

mu_true <- c(-4, 1, 3)
for (k in 1:3) {
  mu_name <- paste0('mu[', k, ']')
  util$plot_expectand_pushforward(samples[[mu_name]], 25,
                                  display_name=mu_name,
                                  baseline=mu_true[k],
                                  baseline_col=util$c_mid_teal)
}
```




```
par(mfrow=c(1, 1), mar=c(0, 0, 0, 0))

plot_simplex_samples(c(samples[['lambda[1]']], recursive=TRUE),
                     c(samples[['lambda[2]']], recursive=TRUE),
                     c(samples[['lambda[3]']], recursive=TRUE),
                     baseline=lambda_true)
```



If the observational model is redundant we can in theory prevent the full Bayesian model from being redundant with an asymmetric prior model. Here let's assume that we have domain expertise that constrains the location of each component models to disjoint intervals,

$$
\begin{aligned}
-6 &\lessapprox \mu_1 \lessapprox -2 \\
-2 &\lessapprox \mu_2 \lessapprox +2 \\
+2 &\lessapprox \mu_3 \lessapprox +6.
\end{aligned}
$$

```
fit <- stan(file="stan_programs/normal_mix2b.stan",
            data=data, seed=8438338,
            warmup=1000, iter=2024, refresh=0)
```

Despite this more informative prior model the parameters of the second and third component models still exhibit split $\hat{R}$ warnings.

```
diagnostics <- util$extract_hmc_diagnostics(fit)
util$check_all_hmc_diagnostics(diagnostics)
```

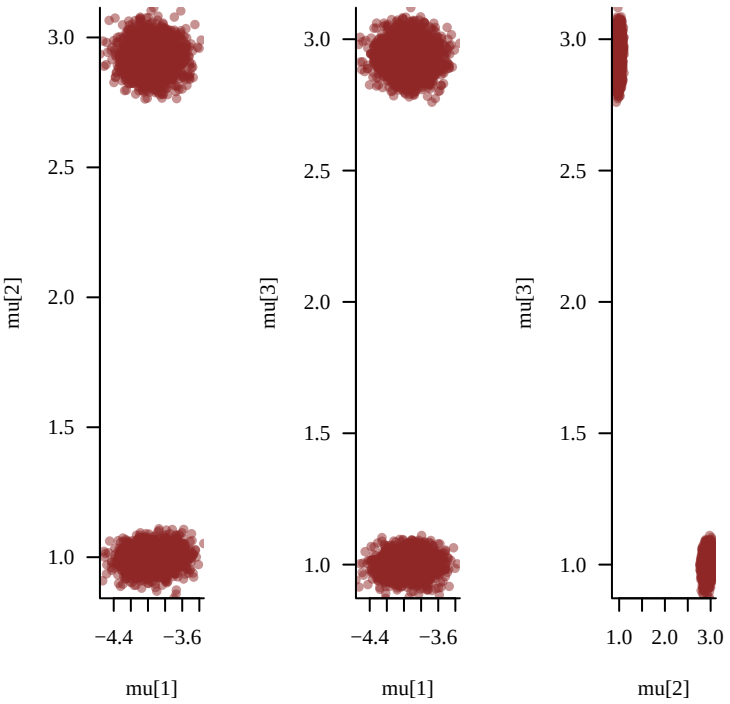
```
  All Hamiltonian Monte Carlo diagnostics are consistent with reliable
Markov chain Monte Carlo.
```

```
samples <- util$extract_expectand_vals(fit)
base_samples <- util$filter_expectands(samples,
                                       c('mu', 'lambda'),
                                       check_arrays=TRUE)
util$check_all_expectand_diagnostics(base_samples)
```

```
mu[2]:
  Split hat{R} (21.965) exceeds 1.1.

mu[3]:
  Split hat{R} (22.389) exceeds 1.1.

lambda[2]:
  Split hat{R} (6.540) exceeds 1.1.

lambda[3]:
  Split hat{R} (6.319) exceeds 1.1.


Split Rhat larger than 1.1 suggests that at least one of the Markov
chains has not reached an equilibrium.
```

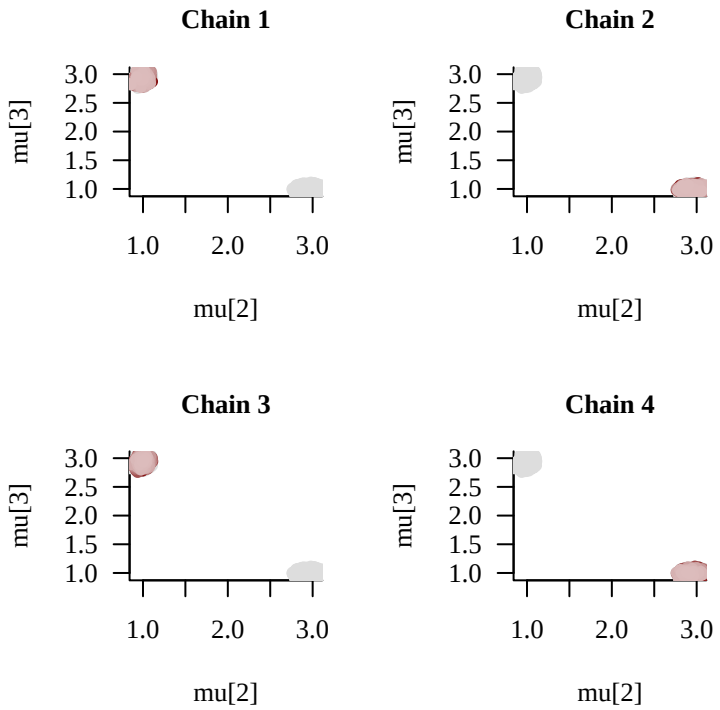
Indeed the multi-modality persists.

```
names <- sapply(1:3, function(k) paste0('mu[', k, ']'))
util$plot_div_pairs(names, names, samples, diagnostics)
```



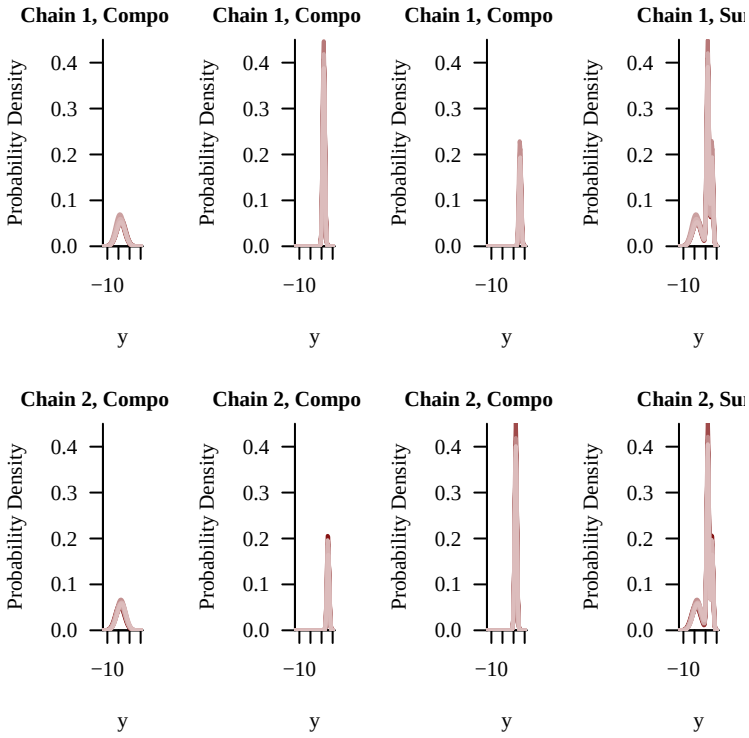
In fact one of the modes concentrates on model configurations with $\mu_3 < \mu_2$, directly contrasting with the behavior of the prior model!

```
util$plot_pairs_by_chain(samples[['mu[2]']], 'mu[2]',
                         samples[['mu[3]']], 'mu[3]')
```



These two modes actually contain model configurations that permute the second and third component models entirely.

```
par(mfrow=c(2, 4), mar=c(5, 5, 2, 1))

for (c in 1:2) {
  for (k in 1:3) {
    plot(NULL, main=paste0('Chain ', c, ', Component ', k),
         xlab="y", ylab="Probability Density",
         xlim=range(xs), ylim=c(0, 0.45))
    plot_component_realizations(k, c)
  }
  plot(NULL, main=paste0('Chain ', c, ', Sum'),
       xlab="y", ylab="Probability Density",
       xlim=range(xs), ylim=c(0, 0.45))
  plot_sum_realizations(c)
}
```



The problem here is that, while strong prior models can suppress redundant or otherwise unwanted modes, they cannot eliminate them entirely. Consequently any modes in the likelihood function will persist into the posterior distribution and potentially trapping any Markov chains that get too close.

One way that we can eliminate the undesired modes is to remove entire regions of the model configuration space. For a mixture of one-dimensional normal models an ordering constraint on the component locations completely removes all but one permutation of the redundant component models. In other words the ordering effectively eliminates the redundancy without limiting the desired flexibility of the model.

```
fit <- stan(file="stan_programs/normal_mix2c.stan",
            data=data, seed=8438338,
            warmup=1000, iter=2024, refresh=0)
```

Encouragingly, all of the diagnostic warnings have ceased.

```
diagnostics <- util$extract_hmc_diagnostics(fit)
util$check_all_hmc_diagnostics(diagnostics)
```

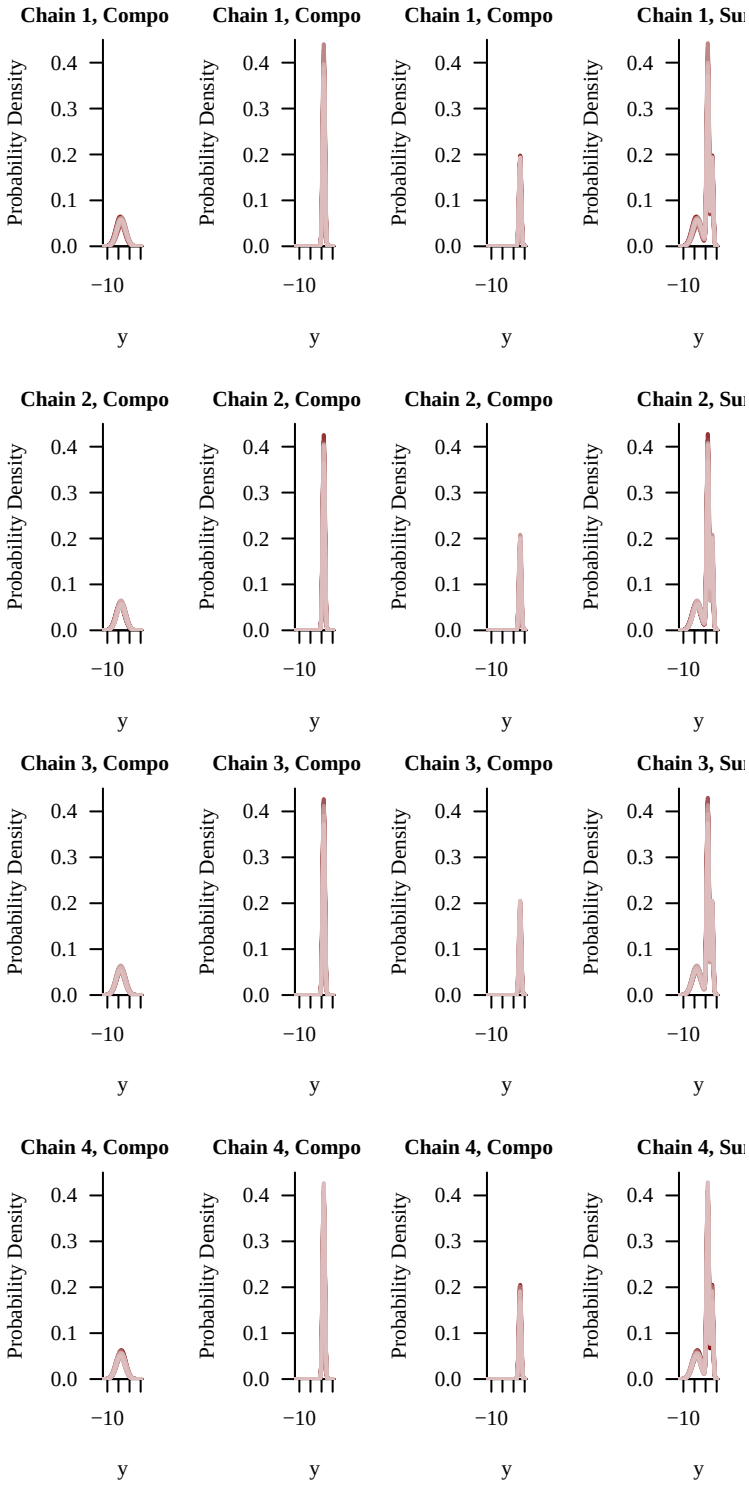
```
  All Hamiltonian Monte Carlo diagnostics are consistent with reliable
Markov chain Monte Carlo.
```

```
samples <- util$extract_expectand_vals(fit)
base_samples <- util$filter_expectands(samples,
                                       c('mu', 'lambda'),
                                       check_arrays=TRUE)
util$check_all_expectand_diagnostics(base_samples)
```

```
All expectands checked appear to be behaving well enough for reliable
Markov chain Monte Carlo estimation.
```

The inferred behavior is consistent across all four Markov chains.

```
par(mfrow=c(2, 4), mar=c(5, 5, 2, 1))

for (c in c(1, 2, 3, 4)) {
  for (k in 1:3) {
    plot(NULL, main=paste0('Chain ', c, ', Component ', k),
         xlab="y", ylab="Probability Density",
         xlim=range(xs), ylim=c(0, 0.45))
    plot_component_realizations(k, c)
  }
  plot(NULL, main=paste0('Chain ', c, ', Sum'),
       xlab="y", ylab="Probability Density",
       xlim=range(xs), ylim=c(0, 0.45))
  plot_sum_realizations(c)
}
```

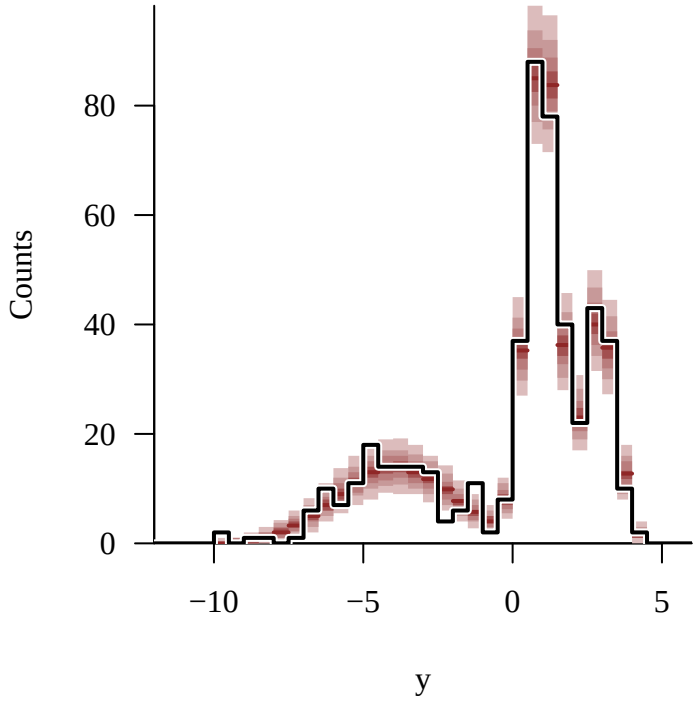


Now that our posterior computation is better behaved we can trust our posterior quantification.

First we check for any retrodictive tension.

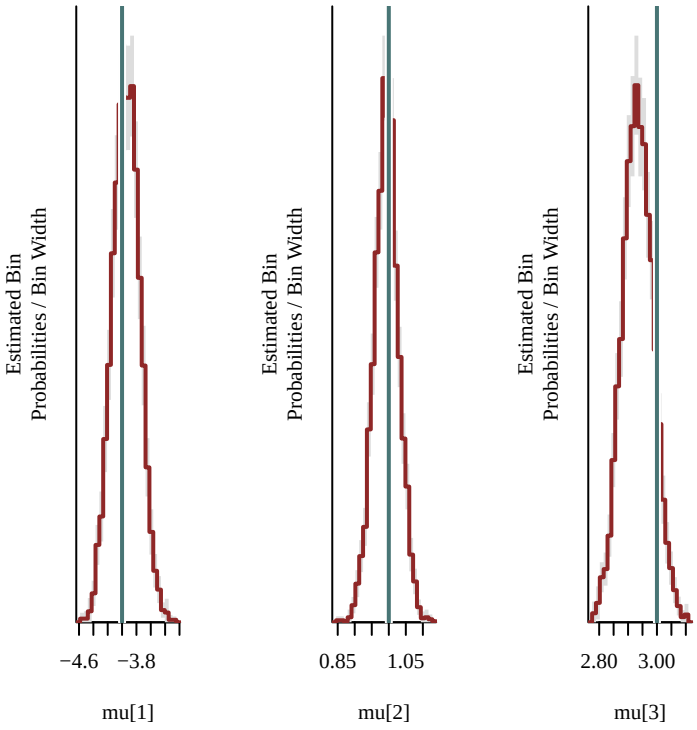
```
par(mfrow=c(1, 1), mar=c(5, 5, 1, 1))

util$plot_hist_quantiles(samples, 'y_pred', -12, 6, 0.5,
                         baseline_values=data$y, xlab="y")
```

```
Warning in check_bin_containment(bin_min, bin_max, collapsed_values,
"predictive value"): 13 predictive values (0.0%) fell below the binning.
```

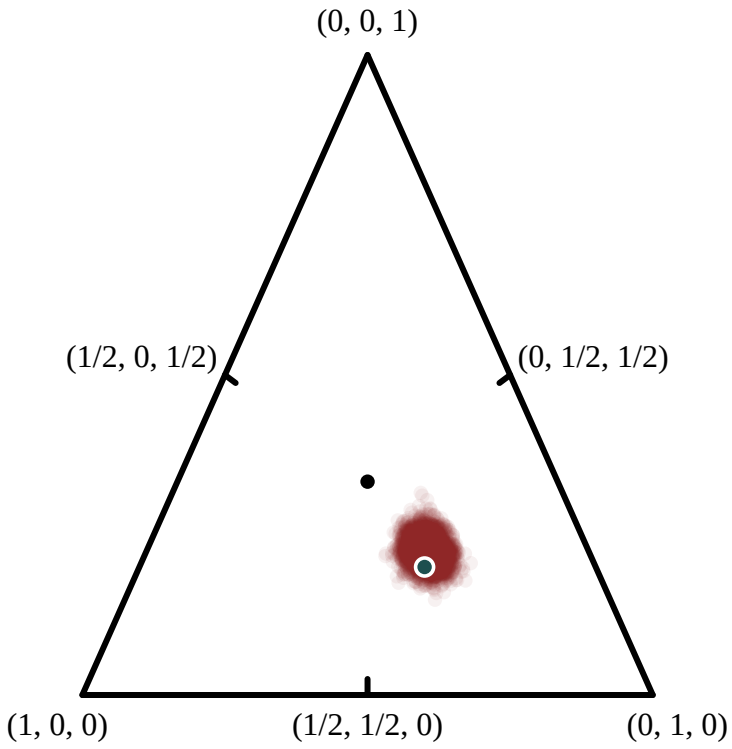


With no signs of model inadequacies we can analyze our posterior inferences. Conveniently they all concentrate around the true behavior of the simulation data generating process.

```
par(mfrow=c(1, 3), mar=c(5, 5, 1, 1))

for (k in 1:3) {
  mu_name <- paste0('mu[', k, ']')
  util$plot_expectand_pushforward(samples[[mu_name]], 25,
                                  display_name=mu_name,
                                  baseline=mu_true[k],
                                  baseline_col=util$c_mid_teal)
}
```




```
par(mfrow=c(1, 1), mar=c(0, 0, 0, 0))

plot_simplex_samples(c(samples[['lambda[1]']], recursive=TRUE),
                     c(samples[['lambda[2]']], recursive=TRUE),
                     c(samples[['lambda[3]']], recursive=TRUE),
                     baseline=lambda_true)
```



5.5.3 Unknown Component Probabilities, Locations, and Scales

Now let's push the analysis even further and try to infer *all* of the component parameters at the same time. In this case nothing distinguishes the component models from each other; the model has reached peak redundancy.

```
fit <- stan(file="stan_programs/normal_mix3a.stan",
            data=data, seed=8438338,
            warmup=1000, iter=2024, refresh=0)
```

The computational diagnostics are now so generous that all of the parameters get a split $\hat{R}$ warning.

```
diagnostics <- util$extract_hmc_diagnostics(fit)
util$check_all_hmc_diagnostics(diagnostics)
```

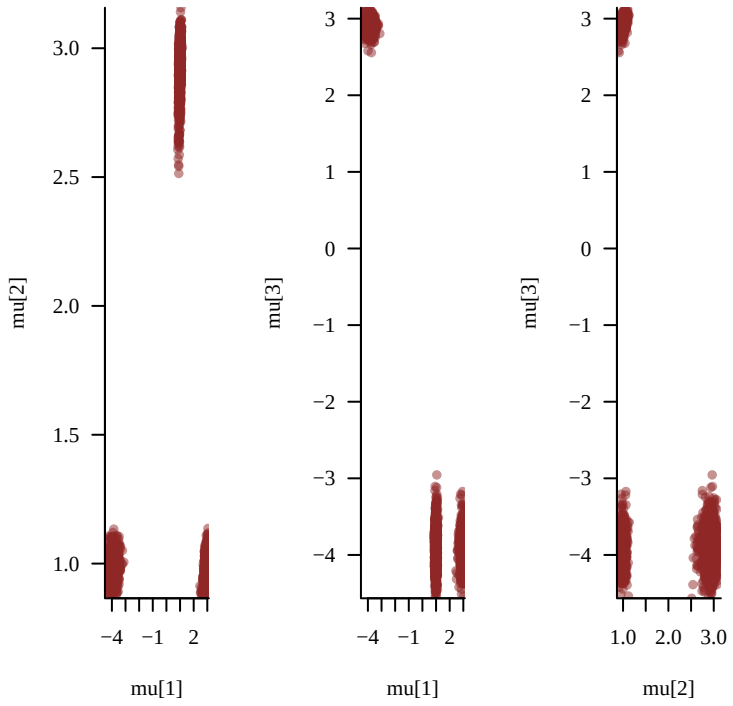
```
  All Hamiltonian Monte Carlo diagnostics are consistent with reliable
Markov chain Monte Carlo.
```

```
samples <- util$extract_expectand_vals(fit)
base_samples <- util$filter_expectands(samples,
                                       c('mu', 'sigma', 'lambda'),
                                       check_arrays=TRUE)
util$check_all_expectand_diagnostics(base_samples)
```

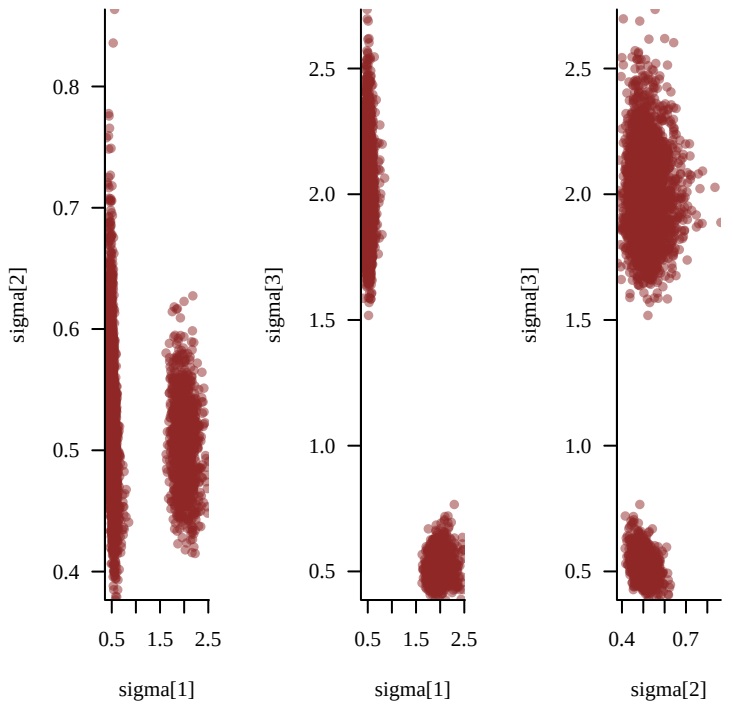
```
mu[1]:
  Split hat{R} (22.613) exceeds 1.1.

mu[2]:
  Split hat{R} (15.986) exceeds 1.1.

mu[3]:
  Split hat{R} (17.349) exceeds 1.1.

sigma[1]:
  Split hat{R} (7.667) exceeds 1.1.

sigma[3]:
  Split hat{R} (4.759) exceeds 1.1.

lambda[1]:
  Split hat{R} (5.073) exceeds 1.1.

lambda[2]:
  Split hat{R} (5.524) exceeds 1.1.

lambda[3]:
  Split hat{R} (1.561) exceeds 1.1.


Split Rhat larger than 1.1 suggests that at least one of the Markov
chains has not reached an equilibrium.
```

Examining some pair plots we can see the horde of modes responsible for the split $\hat{R}$ warnings.

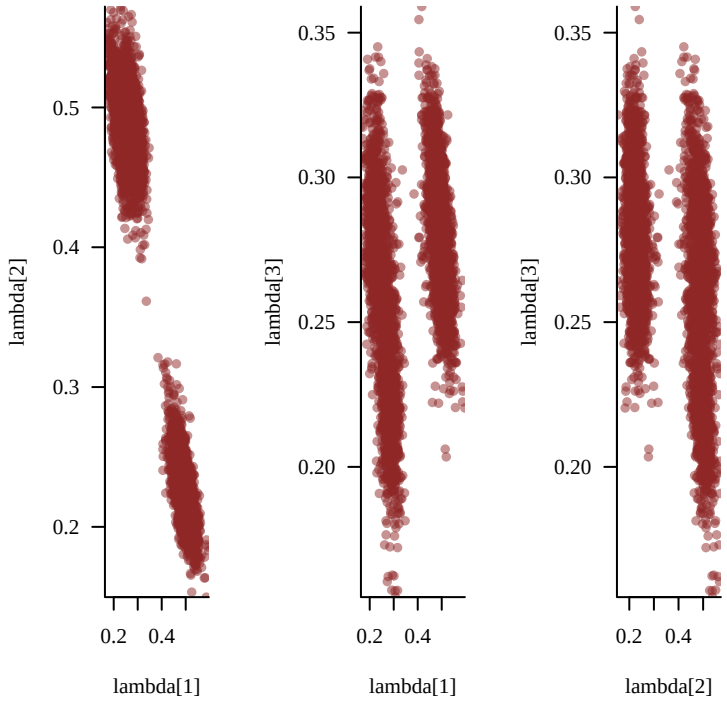
```
names <- sapply(1:3, function(k) paste0('mu[', k, ']'))
util$plot_div_pairs(names, names, samples, diagnostics)
```



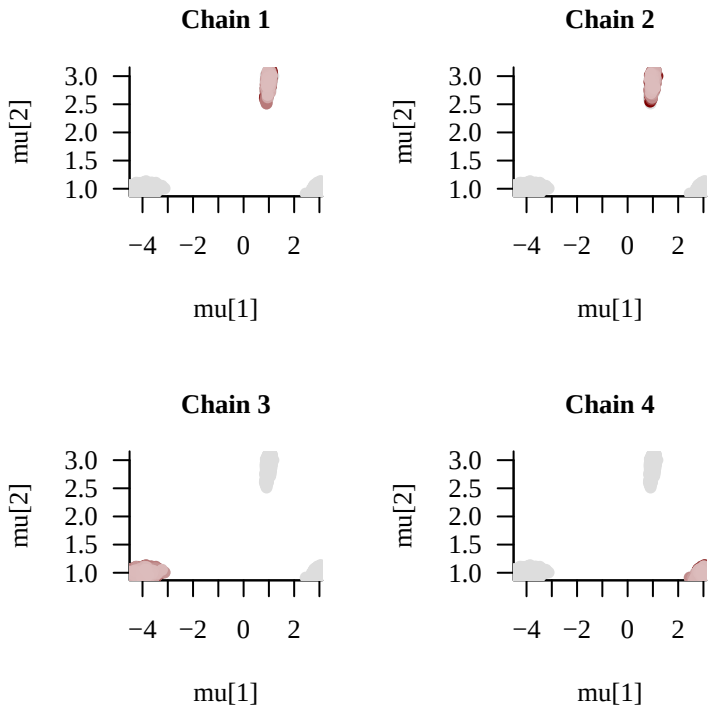

```
names <- sapply(1:3, function(k) paste0('sigma[', k, ']'))
util$plot_div_pairs(names, names, samples, diagnostics)
```



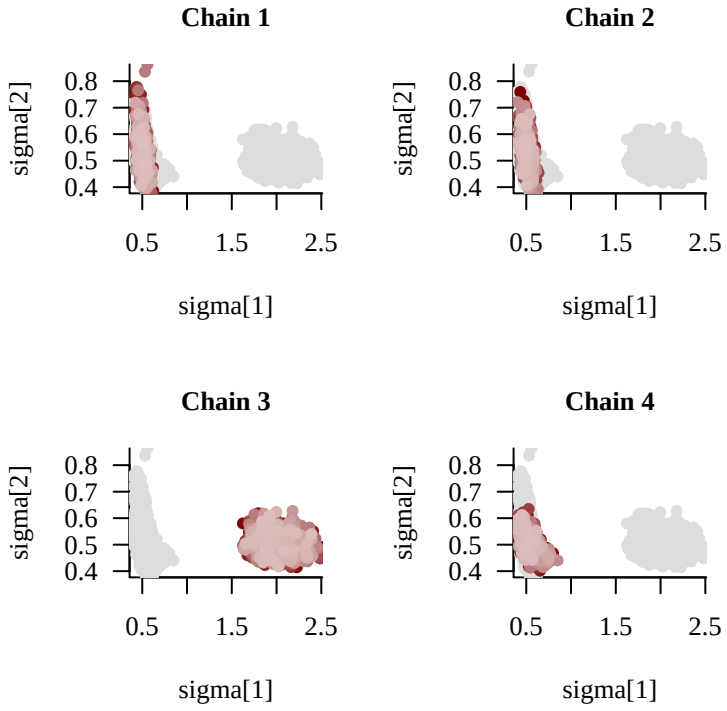
```
names <- sapply(1:3, function(k) paste0('lambda[', k, ']'))
util$plot_div_pairs(names, names, samples, diagnostics)
```



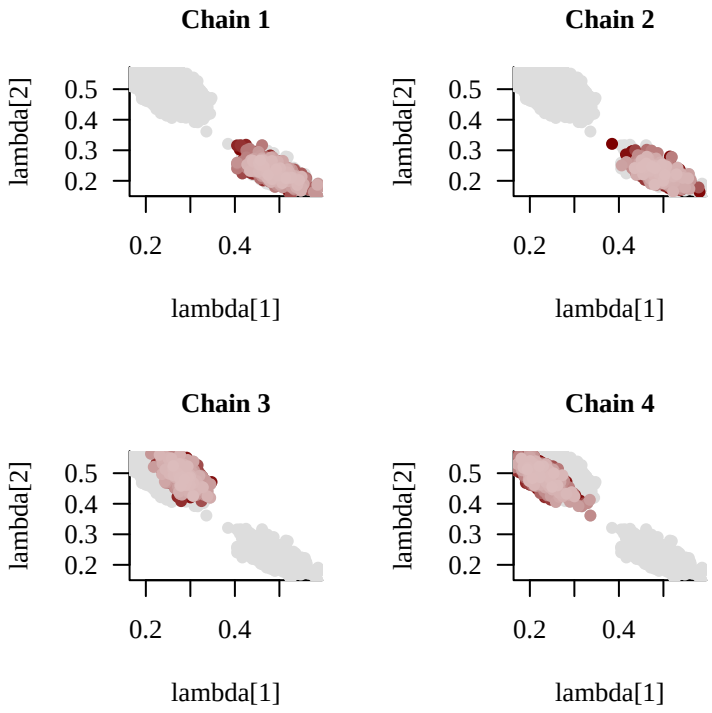

```
util$plot_pairs_by_chain(samples[['mu[1]']], 'mu[1]',
                         samples[['mu[2]']], 'mu[2]')
```




```
util$plot_pairs_by_chain(samples[['sigma[1]']], 'sigma[1]',
                         samples[['sigma[2]']], 'sigma[2]')
```



```
util$plot_pairs_by_chain(samples[['lambda[1]']], 'lambda[1]',
                         samples[['lambda[2]']], 'lambda[2]')
```

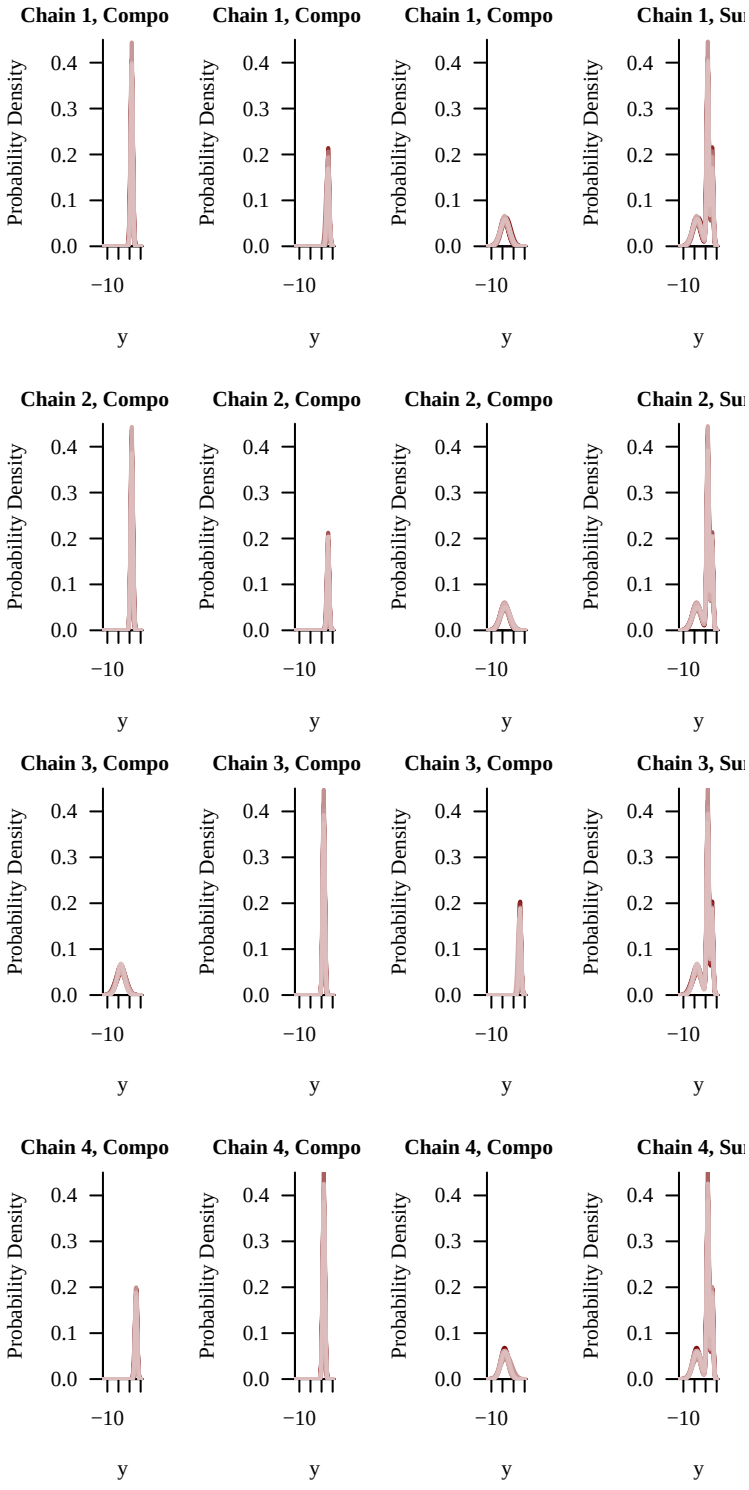


Taking a closer look at the inferred behavior of the component models we can see that all of these modes appear to be permutations of each other.

```
plot_component_realizations <- function(k, c) {
  n <- 1
  for (s in 50 * (1:20)) {
    mu_name <- paste0('mu[', k, ']')
    mu <- samples[[mu_name]][c, s]

    sigma_name <- paste0('sigma[', k, ']')
    sigma <- samples[[sigma_name]][c, s]

    lambda_name <- paste0('lambda[', k, ']')
    lambda <- samples[[lambda_name]][c, s]

    ys <- lambda * dnorm(xs, mu, sigma)
    lines(xs, ys, lwd=2, col=line_colors[n])
    n <- n + 1
  }
```

```
}

plot_sum_realizations <- function(c) {
  n <- 1
  for (s in 50 * (1:20)) {
    mu_names <- sapply(1:3, function(k) paste0('mu[', k, ']'))
    mu <- sapply(mu_names, function(name) samples[[name]][c, s])

    sigma_names <- sapply(1:3, function(k) paste0('sigma[', k, ']'))
    sigma <- sapply(sigma_names, function(name) samples[[name]][c, s])

    lambda_names <- sapply(1:3, function(k) paste0('lambda[', k, ']'))
    lambda <- sapply(lambda_names, function(name) samples[[name]][c, s])

    ys <- rep(0, length(xs))
    for (k in 1:3) {
      ys <- ys + lambda[k] * dnorm(xs, mu[k], sigma[k])
    }
    lines(xs, ys, lwd=2, col=line_colors[n])
    n <- n + 1
  }
}
```

```
par(mfrow=c(2, 4), mar=c(5, 5, 2, 1))

for (c in 1:4) {
  for (k in 1:3) {
    plot(NULL, main=paste0('Chain ', c, ', Component ', k),
         xlab="y", ylab="Probability Density",
         xlim=range(xs), ylim=c(0, 0.45))
    plot_component_realizations(k, c)
  }
  plot(NULL, main=paste0('Chain ', c, ', Sum'),
       xlab="y", ylab="Probability Density",
       xlim=range(xs), ylim=c(0, 0.45))
  plot_sum_realizations(c)
}
```



Can breaking the redundancy with an ordering on the component locations resolve some of

this degeneracy?

```
fit <- stan(file="stan_programs/normal_mix3b.stan",
            data=data, seed=8438338,
            warmup=1000, iter=2024, refresh=0)
```

We don't see quite as many split $\hat{R}$ warnings, so perhaps we've made some progress.

```
diagnostics <- util$extract_hmc_diagnostics(fit)
util$check_all_hmc_diagnostics(diagnostics)
```

```
  All Hamiltonian Monte Carlo diagnostics are consistent with reliable
Markov chain Monte Carlo.
```

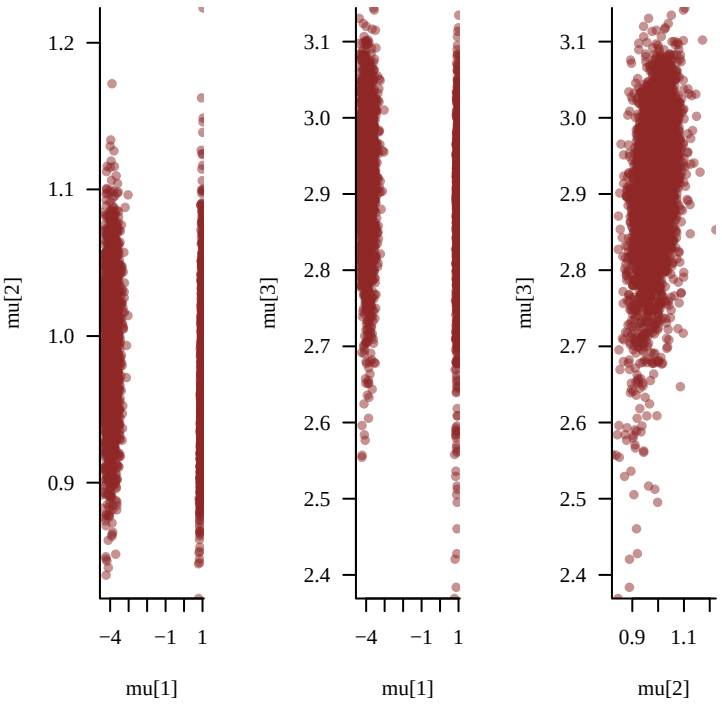
```
samples <- util$extract_expectand_vals(fit)
base_samples <- util$filter_expectands(samples,
                                       c('mu', 'sigma', 'lambda'),
                                       check_arrays=TRUE)
util$check_all_expectand_diagnostics(base_samples)
```

```
mu[1]:
  Split hat{R} (12.694) exceeds 1.1.

sigma[1]:
  Split hat{R} (5.107) exceeds 1.1.

sigma[2]:
  Split hat{R} (14.117) exceeds 1.1.

sigma[3]:
  Split hat{R} (1.119) exceeds 1.1.

lambda[1]:
  Split hat{R} (2.603) exceeds 1.1.

lambda[2]:
  Split hat{R} (1.310) exceeds 1.1.

lambda[3]:
  Split hat{R} (1.701) exceeds 1.1.
```

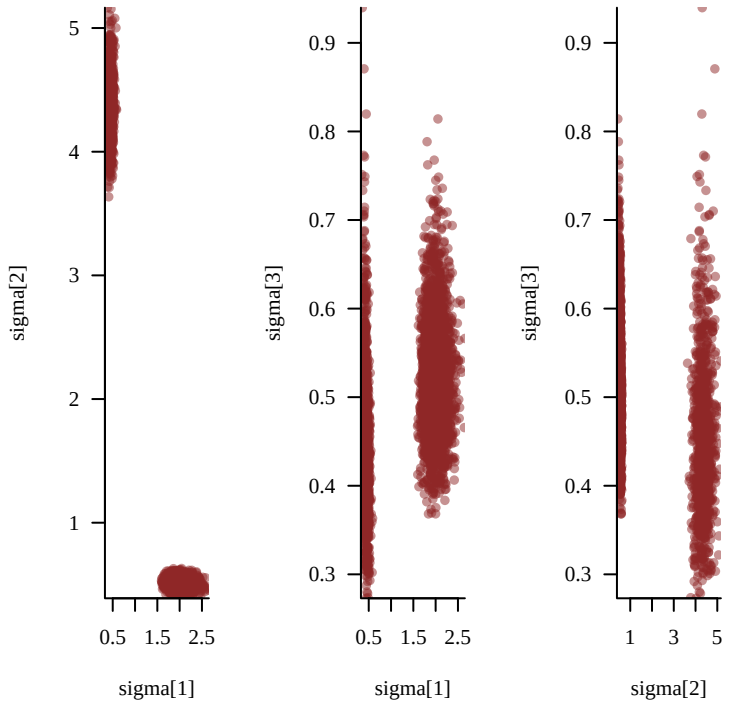
```
Split Rhat larger than 1.1 suggests that at least one of the Markov
chains has not reached an equilibrium.
```

Indeed it looks like we may be down to just two modes.

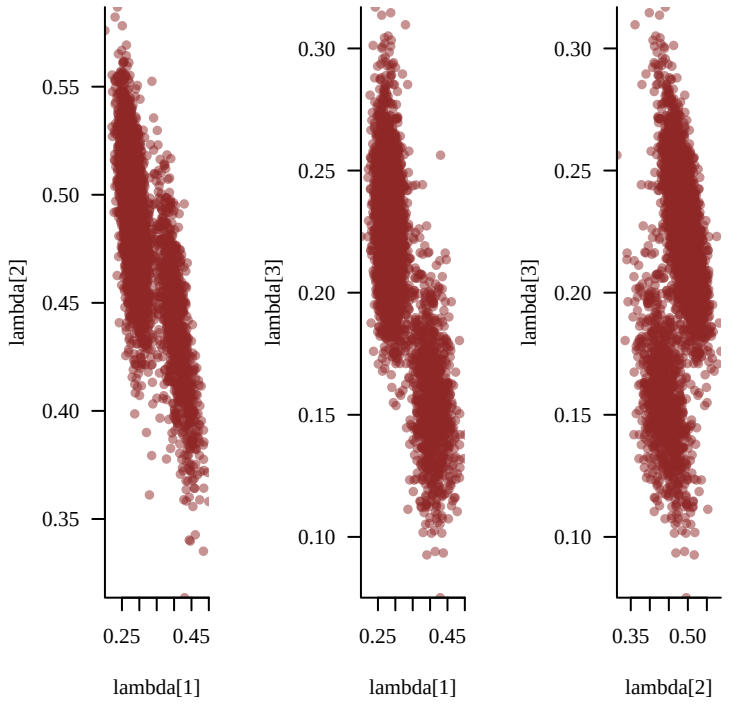
```
names <- sapply(1:3, function(k) paste0('mu[', k, ']'))
util$plot_div_pairs(names, names, samples, diagnostics)
```




```
names <- sapply(1:3, function(k) paste0('sigma[', k, ']'))
util$plot_div_pairs(names, names, samples, diagnostics)
```

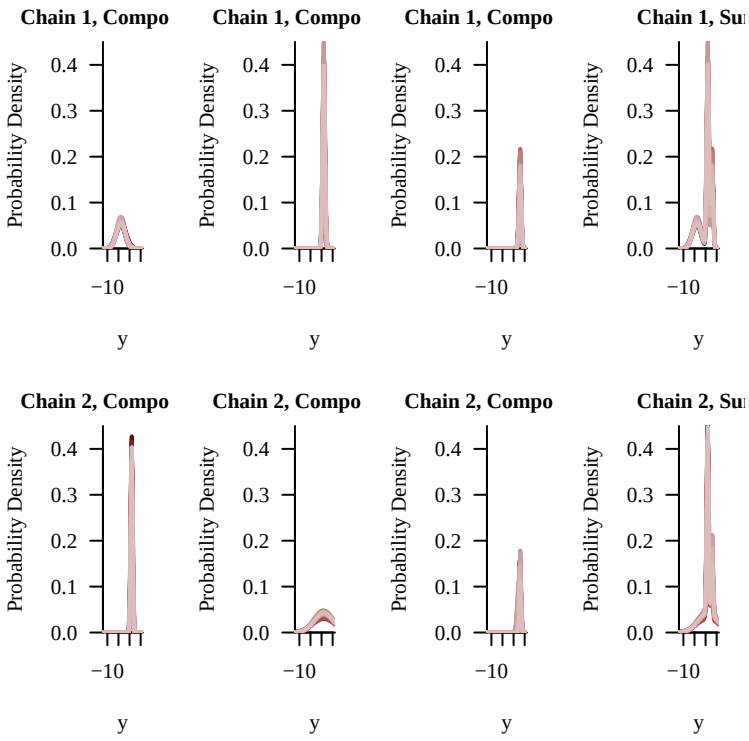


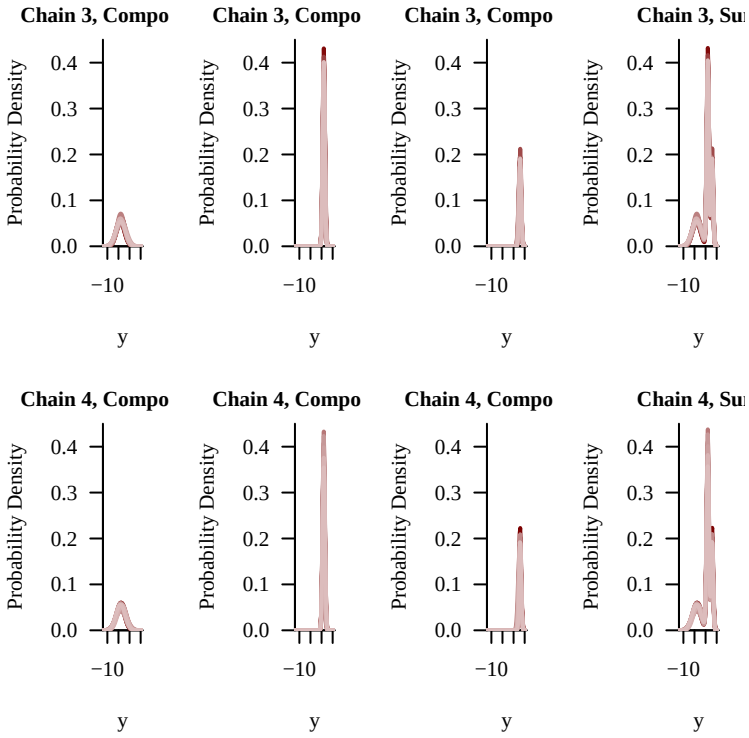

```
names <- sapply(1:3, function(k) paste0('lambda[', k, ']'))
util$plot_div_pairs(names, names, samples, diagnostics)
```



Investigating the inferred component behaviors more closely reveals something interesting.

```
par(mfrow=c(2, 4), mar=c(5, 5, 2, 1))

for (c in 1:4) {
  for (k in 1:3) {
    plot(NULL, main=paste0('Chain ', c, ', Component ', k),
         xlab="y", ylab="Probability Density",
         xlim=range(xs), ylim=c(0, 0.45))
    plot_component_realizations(k, c)
  }
  plot(NULL, main=paste0('Chain ', c, ', Sum'),
       xlab="y", ylab="Probability Density",
       xlim=range(xs), ylim=c(0, 0.45))
  plot_sum_realizations(c)
}
```





The first, third, and fourth Markov chains exhibit the ideal behavior, with each component matching the behavior of one of the component simulation data generating processes. In the second Markov chain, however, the first component observational model moves up to capture the second component simulation data generating process. Because of the ordering constraint this pushes the second component observational model up as well. The second component observational model then uses the pliability of the component scales to expand and cover the observed data at smaller values that the first component observational model missed.

This latter contortion is less consistent with the observed data, which we can see in the corresponding retrodictive checks.

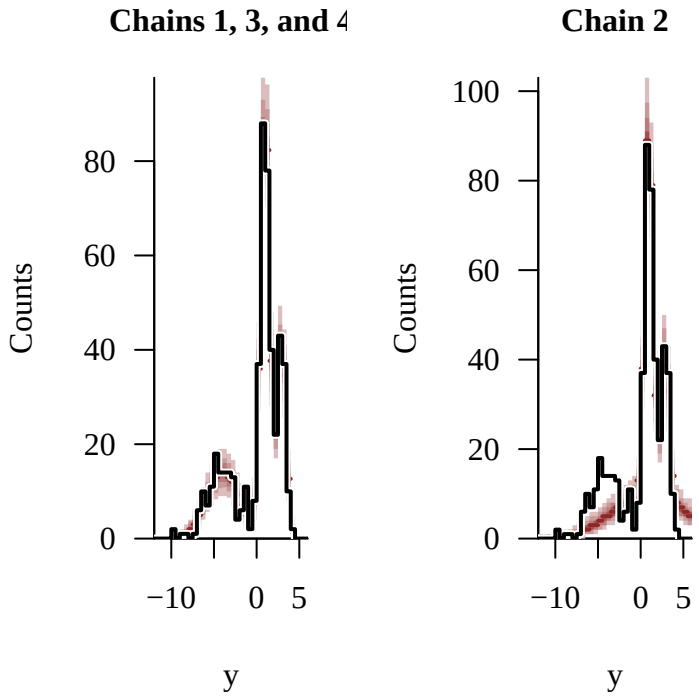
```
par(mfrow=c(1, 2), mar=c(5, 5, 3, 1))

samples134 <- lapply(samples, function(s) s[c(1, 3, 4),])
util$plot_hist_quantiles(samples134, 'y_pred', -12, 6, 0.5,
                         baseline_values=data$y, xlab="y",
                         main="Chains 1, 3, and 4")
```

```
Warning in check_bin_containment(bin_min, bin_max, collapsed_values,
"predictive value"): 16 predictive values (0.0%) fell below the binning.
```

```
samples2 <- lapply(samples, function(s) array(s[2,], dim=c(1, 1024)))
util$plot_hist_quantiles(samples2, 'y_pred', -12, 6, 0.5,
                         baseline_values=data$y, xlab="y",
                         main="Chain 2")
```

```
Warning in check_bin_containment(bin_min, bin_max, collapsed_values,
"predictive value"): 354 predictive values (0.1%) fell below the binning.
```

```
Warning in check_bin_containment(bin_min, bin_max, collapsed_values,
"predictive value"): 27805 predictive values (5.4%) fell above the binning.
```



The exact posterior distribution will allocate less probability to the second mode, allowing the behaviors in the first mode to dominate our inferences. Unfortunately Markov chain Monte Carlo cannot reliably estimate the relative probabilities of the two modes, and in practice we don't know how much to discount the second mode.

5.5.4 Unknown Number of Components

Finally let's see what happens when we use a mixture observational model with more components than are actually in the simulation data generating process. Note that we're starting

with an ordering constraint to make the posterior distribution as well-behaved as possible from the beginning.

```
data$K <- 5

fit <- stan(file="stan_programs/normal_mix4.stan",
            data=data, seed=8438338,
            warmup=1000, iter=2024, refresh=0)
```

We have good news and we have bad news. The good news is that there are almost no split $\hat{R}$ warnings; the bad news is that just about every other warning has triggered. In particular the tree depth saturation and small empirical effective sample size warnings suggest strong Markov chain autocorrelations which themselves hint at complex posterior uncertainties.

```
diagnostics <- util$extract_hmc_diagnostics(fit)
util$check_all_hmc_diagnostics(diagnostics)
```

```
  Chain 1: 1 of 1024 transitions (0.1%) diverged.
  Chain 1: 52 of 1024 transitions (5.08%)
           saturated the maximum treedepth of 10.

  Chain 2: 36 of 1024 transitions (3.52%)
           saturated the maximum treedepth of 10.

  Chain 3: 116 of 1024 transitions (11.33%)
           saturated the maximum treedepth of 10.

  Chain 4: 182 of 1024 transitions (17.77%)
           saturated the maximum treedepth of 10.

  Divergent Hamiltonian transitions result from unstable numerical
trajectories.  These instabilities are often due to degenerate target
geometry, especially "pinches".  If there are only a small number of
divergences then running with adept_delta larger than 0.801 may reduce
the instabilities at the cost of more expensive Hamiltonian
transitions.

  Numerical trajectories that saturate the maximum treedepth have
terminated prematurely.  Increasing max_depth above 10 should result in
more expensive, but more efficient, Hamiltonian transitions.
```

```
samples <- util$extract_expectand_vals(fit)
base_samples <- util$filter_expectands(samples,
                                       c('mu', 'sigma', 'lambda'),
                                       check_arrays=TRUE)
util$check_all_expectand_diagnostics(base_samples)
```

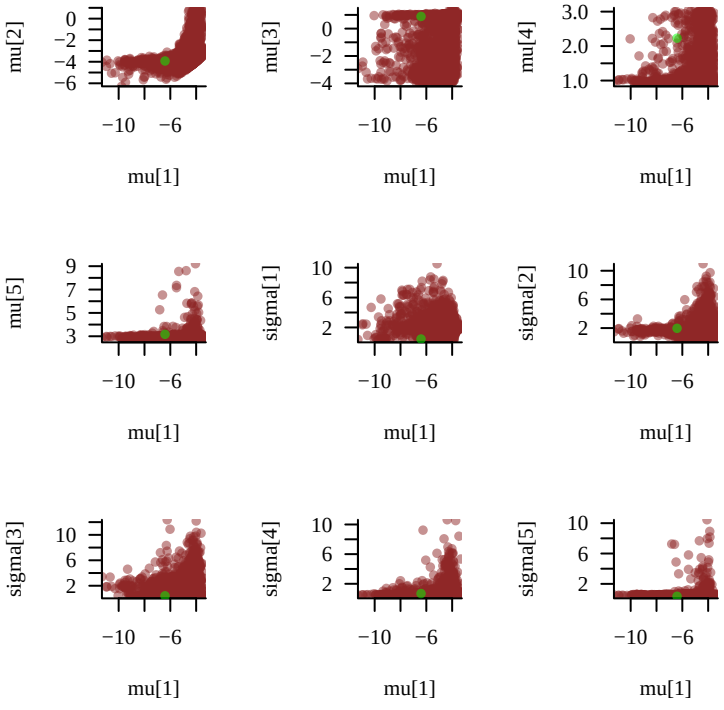
```
mu[1]:
  Chain 1: Left tail hat{xi} (0.409) exceeds 0.25.
  Chain 3: Left tail hat{xi} (0.320) exceeds 0.25.
  Chain 2: hat{ESS} (92.084) is smaller than desired (100).
  Chain 3: hat{ESS} (86.435) is smaller than desired (100).

mu[2]:
  Chain 3: hat{ESS} (84.762) is smaller than desired (100).

mu[3]:
  Chain 1: hat{ESS} (30.790) is smaller than desired (100).
  Chain 2: hat{ESS} (47.728) is smaller than desired (100).
  Chain 3: hat{ESS} (34.329) is smaller than desired (100).
  Chain 4: hat{ESS} (30.072) is smaller than desired (100).

mu[4]:
  Chain 2: Right tail hat{xi} (0.484) exceeds 0.25.
  Chain 3: Right tail hat{xi} (1.429) exceeds 0.25.
  Chain 1: hat{ESS} (27.742) is smaller than desired (100).
  Chain 2: hat{ESS} (32.215) is smaller than desired (100).
  Chain 3: hat{ESS} (29.548) is smaller than desired (100).
  Chain 4: hat{ESS} (26.779) is smaller than desired (100).

mu[5]:
  Chain 1: Right tail hat{xi} (0.797) exceeds 0.25.
  Chain 4: Right tail hat{xi} (0.383) exceeds 0.25.
  Chain 4: hat{ESS} (87.885) is smaller than desired (100).

sigma[1]:
  Chain 1: Right tail hat{xi} (0.281) exceeds 0.25.
  Chain 2: Right tail hat{xi} (0.339) exceeds 0.25.
  Chain 3: Right tail hat{xi} (0.311) exceeds 0.25.
  Chain 4: Right tail hat{xi} (0.329) exceeds 0.25.

sigma[3]:
  Chain 1: hat{ESS} (58.160) is smaller than desired (100).
```

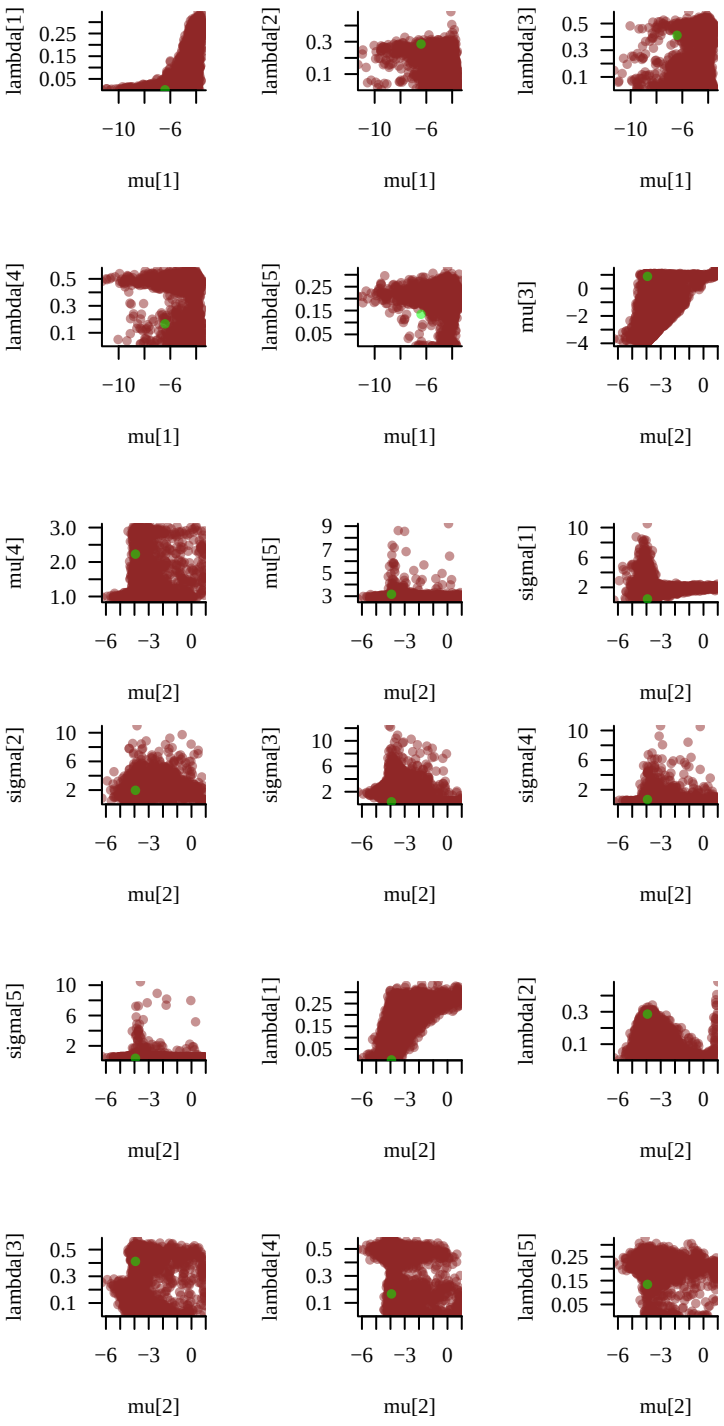
```
  Chain 3: hat{ESS} (52.130) is smaller than desired (100).
  Chain 4: hat{ESS} (40.478) is smaller than desired (100).

sigma[4]:
  Chain 1: Right tail hat{xi} (0.626) exceeds 0.25.
  Chain 2: Right tail hat{xi} (0.602) exceeds 0.25.
  Chain 3: Right tail hat{xi} (1.218) exceeds 0.25.
  Chain 4: Right tail hat{xi} (1.221) exceeds 0.25.
  Chain 2: hat{ESS} (42.709) is smaller than desired (100).
  Chain 3: hat{ESS} (52.978) is smaller than desired (100).

sigma[5]:
  Chain 1: Right tail hat{xi} (0.947) exceeds 0.25.
  Chain 4: Right tail hat{xi} (0.632) exceeds 0.25.
  Chain 4: hat{ESS} (90.384) is smaller than desired (100).

lambda[1]:
  Chain 3: hat{ESS} (94.205) is smaller than desired (100).

lambda[3]:
  Split hat{R} (1.109) exceeds 1.1.
  Chain 1: hat{ESS} (24.711) is smaller than desired (100).
  Chain 2: hat{ESS} (31.587) is smaller than desired (100).
  Chain 3: hat{ESS} (26.128) is smaller than desired (100).
  Chain 4: hat{ESS} (22.753) is smaller than desired (100).

lambda[4]:
  Chain 2: Left tail hat{xi} (0.675) exceeds 0.25.
  Chain 3: Left tail hat{xi} (0.914) exceeds 0.25.
  Split hat{R} (1.113) exceeds 1.1.
  Chain 1: hat{ESS} (27.508) is smaller than desired (100).
  Chain 2: hat{ESS} (21.237) is smaller than desired (100).
  Chain 3: hat{ESS} (23.639) is smaller than desired (100).
  Chain 4: hat{ESS} (20.943) is smaller than desired (100).

lambda[5]:
  Chain 3: Left tail hat{xi} (0.269) exceeds 0.25.
  Chain 4: Left tail hat{xi} (0.337) exceeds 0.25.
  Chain 1: hat{ESS} (37.481) is smaller than desired (100).
  Chain 3: hat{ESS} (61.483) is smaller than desired (100).
  Chain 4: hat{ESS} (40.355) is smaller than desired (100).
```

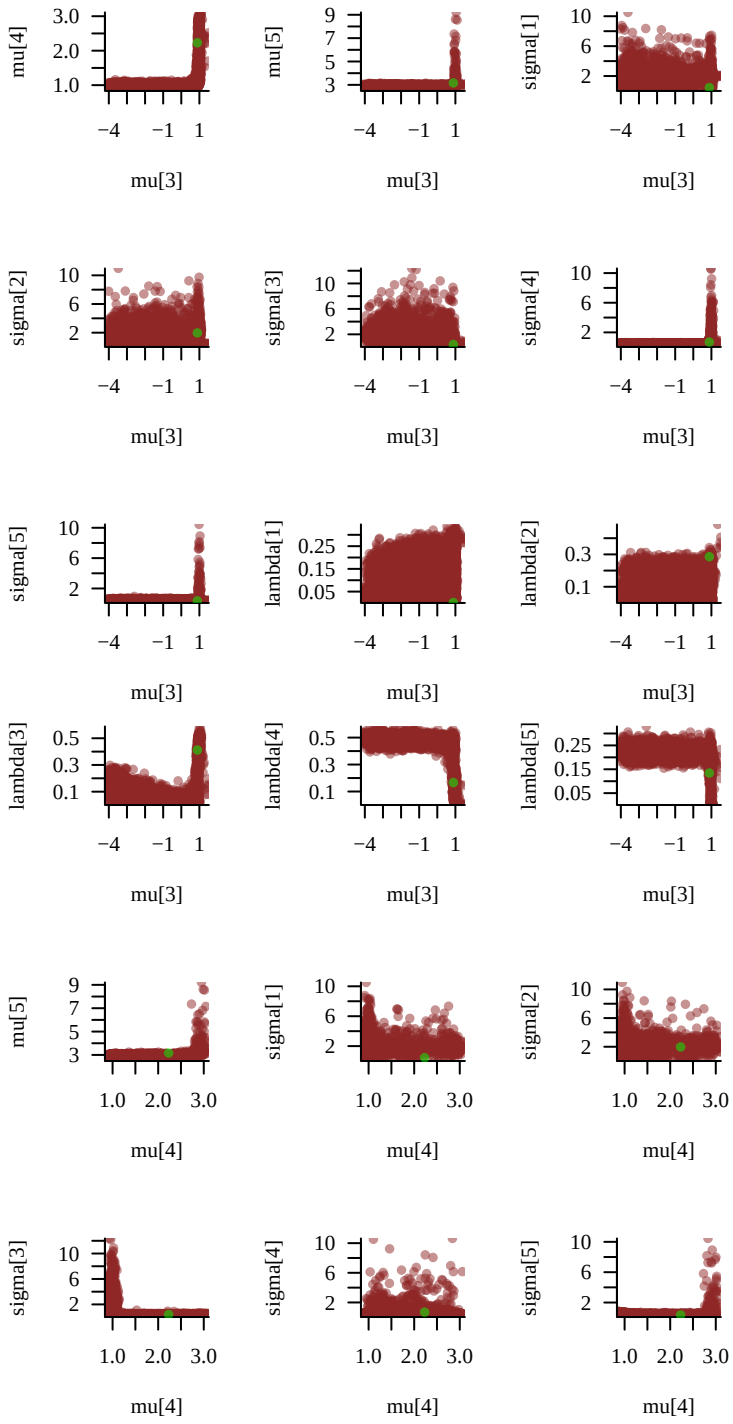
```
Large tail hat{xi}s suggest that the expectand might not be
sufficiently integrable.

Split Rhat larger than 1.1 suggests that at least one of the Markov
chains has not reached an equilibrium.

Small empirical effective sample sizes result in imprecise Markov chain
Monte Carlo estimators.
```

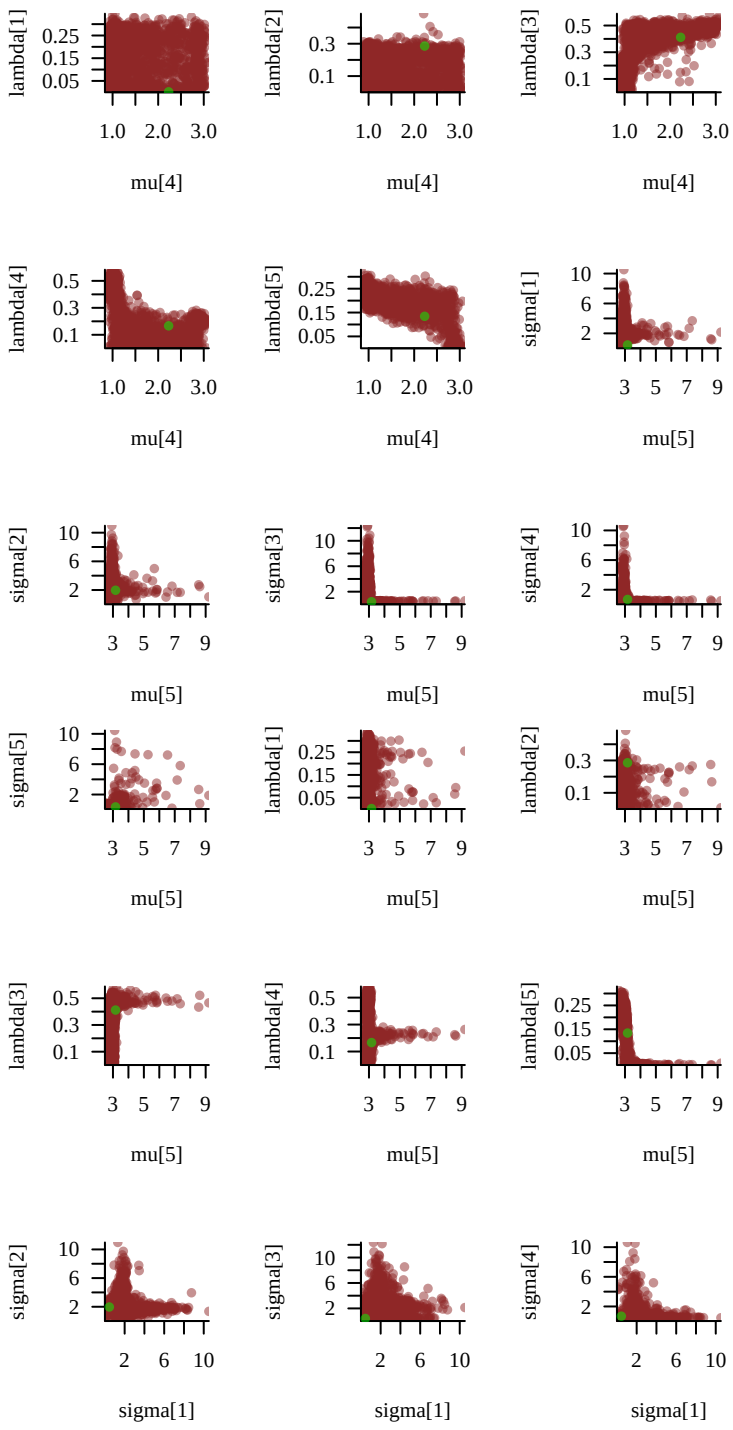
In general I do not recommend looking at every possible pair plot; it is much more productive to use the model structure to prioritize a reasonable number of pair plots. In this case, however, I want to show all of the pair plots just to demonstrate how degenerate the posterior distribution has become here. It truly is the stuff of statistical nightmares.

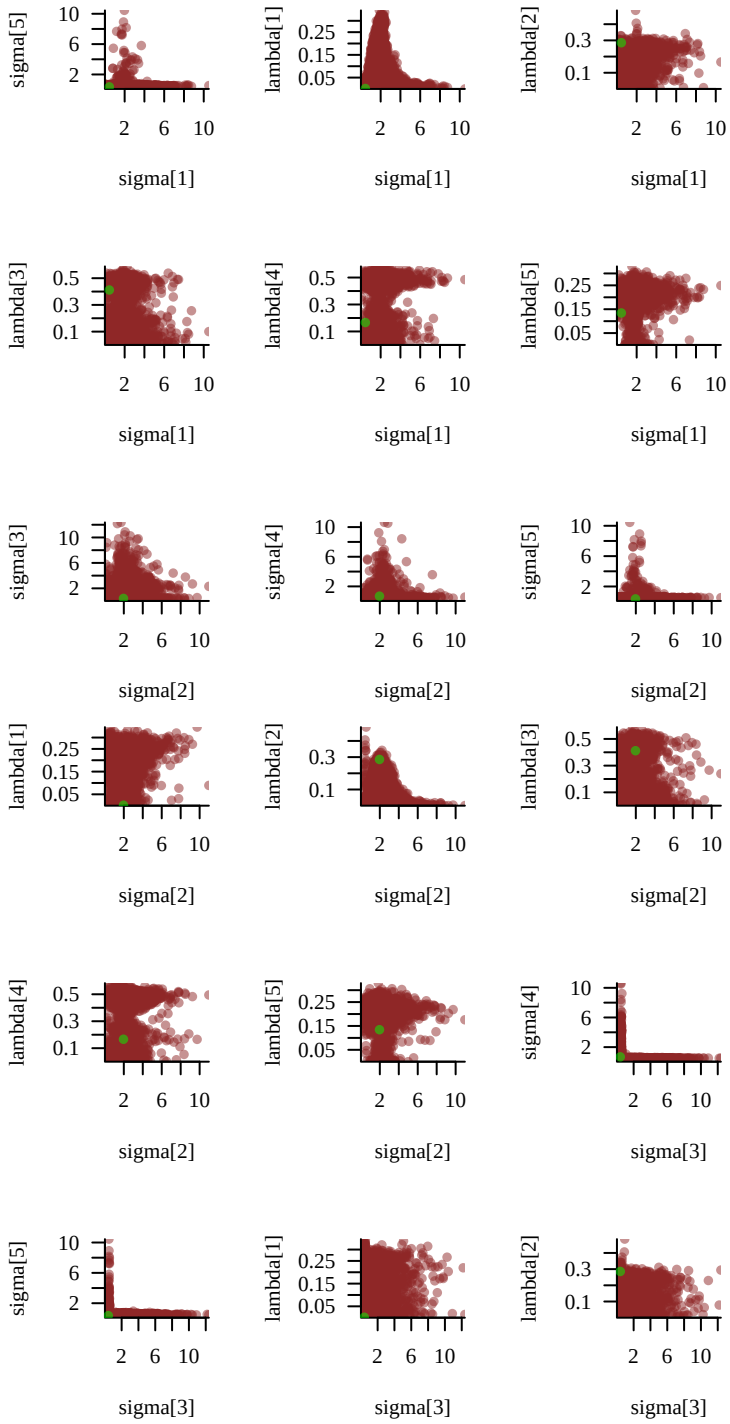
```
names <- c(sapply(c('mu', 'sigma', 'lambda'), function(name)
           sapply(1:5, function(k) paste0(name, '[', k, ']'))))
util$plot_div_pairs(names, names, samples, diagnostics)
```

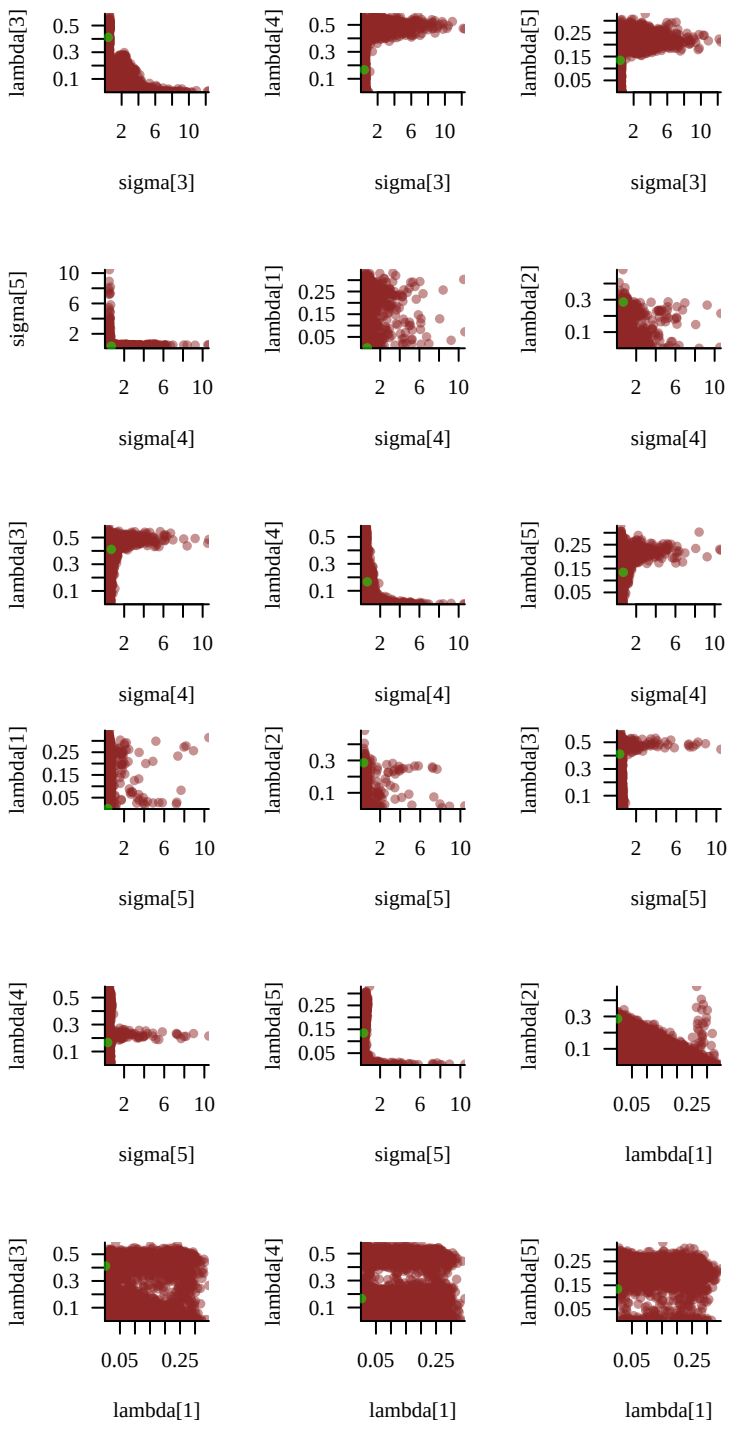


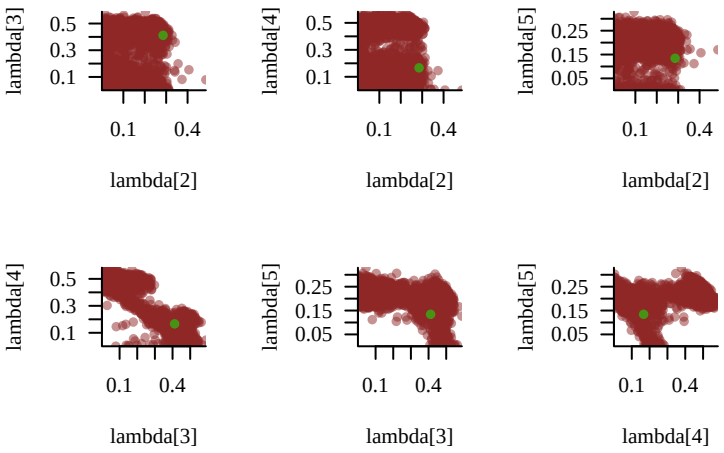

lambda[1]
lambda[2]
lambda[3]
lambda[4]
lambda[5]
mu[3]
mu[4]
mu[5]
sigma[1]
sigma[2]
sigma[3]
sigma[4]
sigma[5]
mu[1]
mu[2]


mu[4]
mu[5]
sigma[1]
sigma[2]
sigma[3]
sigma[4]
sigma[5]
lambda[1]
lambda[2]
lambda[3]
lambda[4]
lambda[5]
mu[3]
mu[5]
sigma[1]
sigma[2]
sigma[3]
sigma[4]
sigma[5]
mu[4]


lambda[1]
lambda[2]
lambda[3]
lambda[4]
lambda[5]
sigma[1]
sigma[2]
sigma[3]
sigma[4]
sigma[5]
mu[4]
mu[5]
sigma[1]

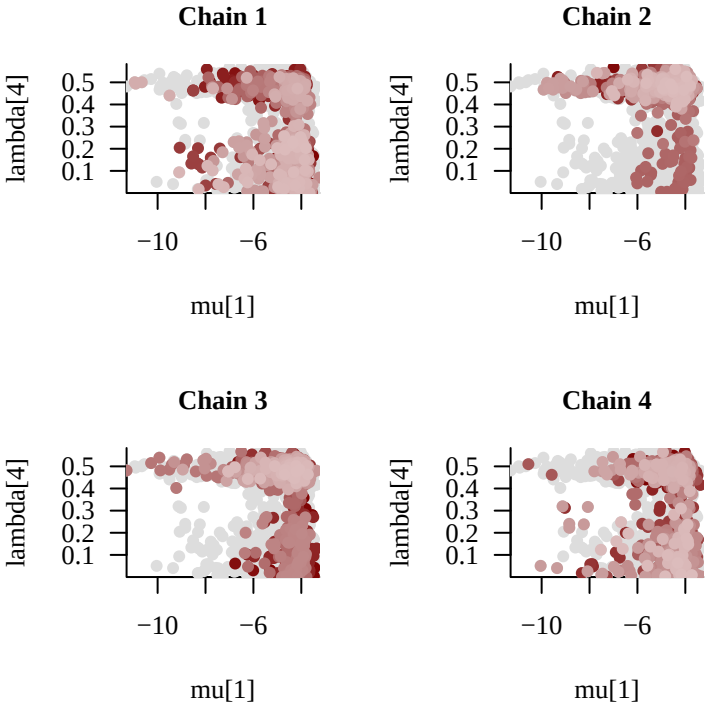

sigma[5]
lambda[1]
lambda[2]
lambda[3]
lambda[4]
lambda[5]
sigma[3]
sigma[4]
sigma[5]
lambda[1]
lambda[2]
lambda[3]
lambda[4]
lambda[5]
sigma[4]
sigma[5]
lambda[1]
lambda[2]
sigma[1]
sigma[2]
sigma[3]


lambda[3]
lambda[4]
lambda[5]
sigma[3]
sigma[5]
lambda[1]
lambda[2]
sigma[4]
lambda[3]
lambda[4]
lambda[5]
sigma[4]
lambda[1]
lambda[2]
lambda[3]
sigma[5]
lambda[4]
lambda[5]
lambda[2]
sigma[5]
lambda[1]
lambda[3]
lambda[4]
lambda[5]
lambda[1]



Interestingly there don't appear to be any isolated modes. To be clear we do see some modal structure, for example in the plot of `mu[1]` against `lambda[4]`, but the individual Markov chains are largely able to transition between the modes without issue. Consequently our Markov chain Monte Carlo estimators are reasonably reliable.

```
util$plot_pairs_by_chain(samples[['mu[1]']], 'mu[1]',
                         samples[['lambda[4]']], 'lambda[4]')
```

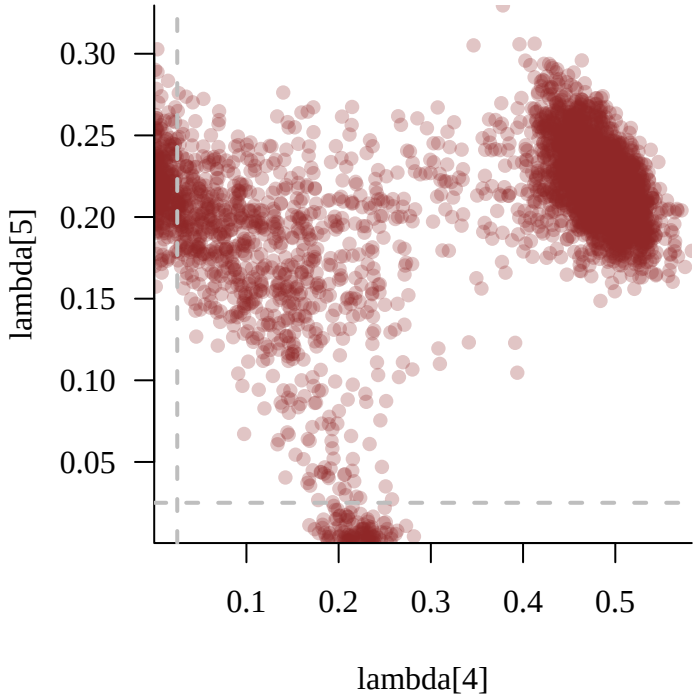


To better understand all of these intricate degeneracies let's take a look at the underlying mixture observational model and reason about the many different ways that it can contort itself

while maintaining consistency with the observed data. Ultimately a mixture observational model with too many components needs to find a way to hide, if not outright eliminate, the contribution from the extraneous components.

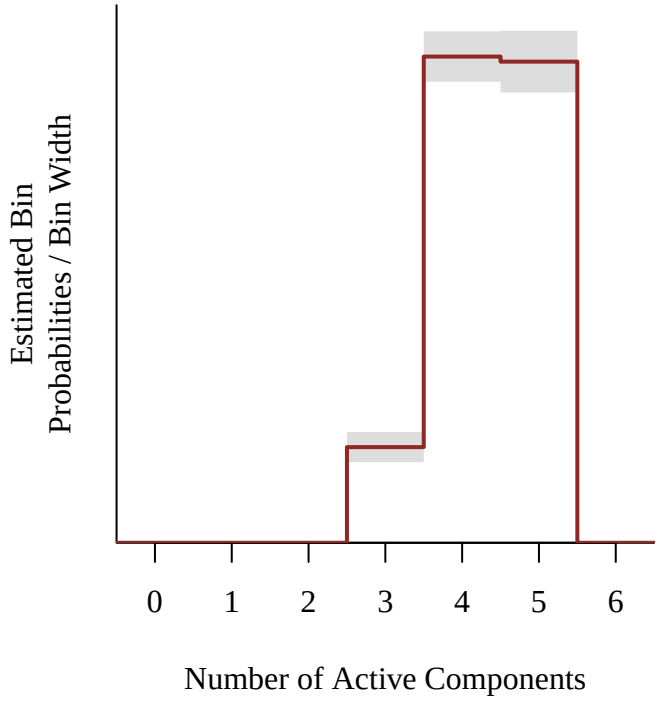
For example the contribution from some of the component models can be "turned off" if the corresponding component probabilities are close to zero. Indeed we can see bands of small component probabilities in the pair plots.

```
par(mfrow=c(1, 1), mar=c(5, 5, 1, 1))

plot(c(samples[['lambda[4]']], recursive=TRUE),
     c(samples[['lambda[5]']], recursive=TRUE),
     pch=16, col="#8F272744",
     xlab='lambda[4]', ylab='lambda[5]')
abline(h=0.025, col='gray', lty=2, lwd=2)
abline(v=0.025, col='gray', lty=2, lwd=2)
```



Let's use the structure of these bands to define a heuristic cut-off between the "active" components that substantially contribute to the overall mixture probability distribution and the "inactive" components that offer only negligible contributions. This cut-off then allows us to approximately count the number of active components in each model configuration.

```
var_repl <- list('lambda'=array(sapply(1:5,
                                       function(k) paste0("lambda[", k, "]"))))
active_components <-
  util$eval_expectand_pushforward(samples,
                                  function(lambda) sum(lambda > 0.025),
                                  var_repl)

par(mfrow=c(1, 1), mar=c(5, 5, 1, 1))

util$plot_expectand_pushforward(active_components,
                                7, flim=c(-0.5, 6.5),
                                display_name="Number of Active Components")
```



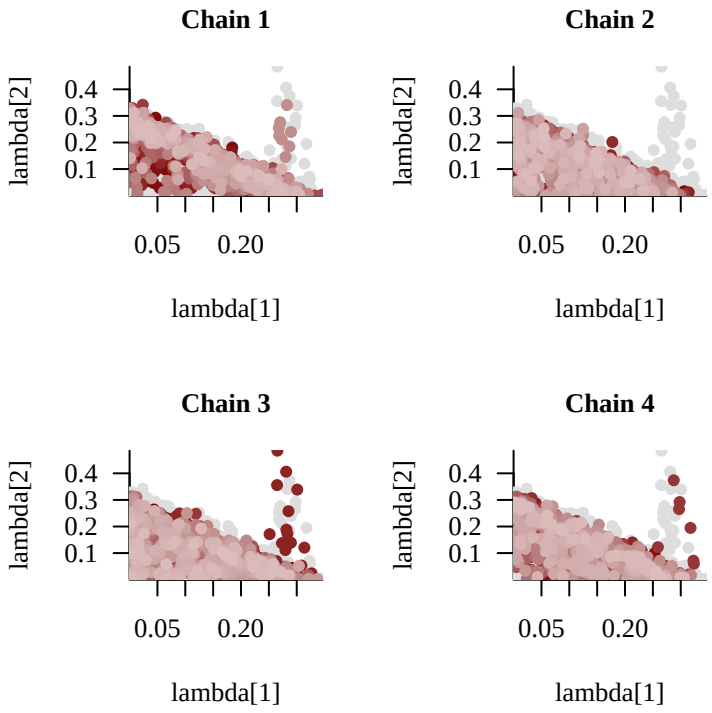
The posterior distribution concentrates on model configurations with three, four, or five active components. Does this behavior align with our understanding of the model and the simulation data generating process?

Well the suppression of model configurations with only one or two active components make sense. None of these model configurations are able to capture all three peaks in the observed data.

The model configurations with three active components are perhaps the most intuitive given the three-component structure of the simulation data generating process. Interestingly *which*

two components are turned off is not fixed but rather varies as the Markov chains evolve. For example sometimes `lambda[1]` is close to zero while `lambda[2]` is not and sometimes `lambda[2]` is close to zero while `lambda[1]` is not. Sometimes they're both far enough away from zero for the corresponding component models to contribute to the mixture probability distribution, and sometimes they're both close enough to zero for the corresponding contributions to be negligible.

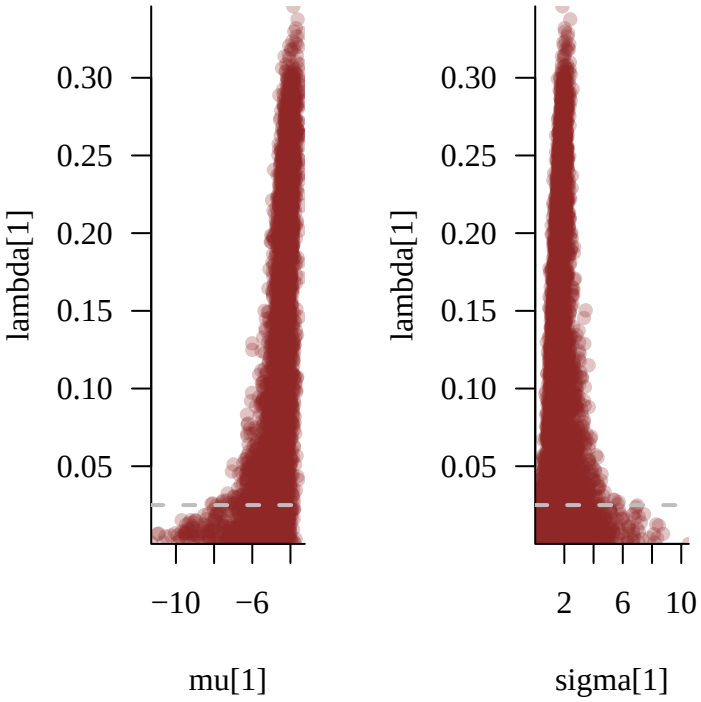
```
util$plot_pairs_by_chain(samples[['lambda[1]']], 'lambda[1]',
                         samples[['lambda[2]']], 'lambda[2]')
```



Note also that when a component becomes inactive the corresponding location and scale parameters are no longer informed by the observed data and their posterior behaviors relax back to the prior constraints. This explains some of the ridges that we see in the pair plots.

```
par(mfrow=c(1, 2), mar=c(5, 5, 1, 1))

plot(c(samples[['mu[1]']], recursive=TRUE),
     c(samples[['lambda[1]']], recursive=TRUE),
     pch=16, col="#8F272744",
     xlab='mu[1]', ylab='lambda[1]')
abline(h=0.025, col='gray', lty=2, lwd=2)
```

```
plot(c(samples[['sigma[1]']], recursive=TRUE),
     c(samples[['lambda[1]']], recursive=TRUE),
     pch=16, col="#8F272744",
     xlab='sigma[1]', ylab='lambda[1]')
abline(h=0.025, col='gray', lty=2, lwd=2)
```



Now the posterior distribution mostly concentrates on model configurations that feature not three but rather four if not five active components. In these cases neighboring components need to collapse against each other so that their sum reconstructs one of the true components in the simulation data generating process.

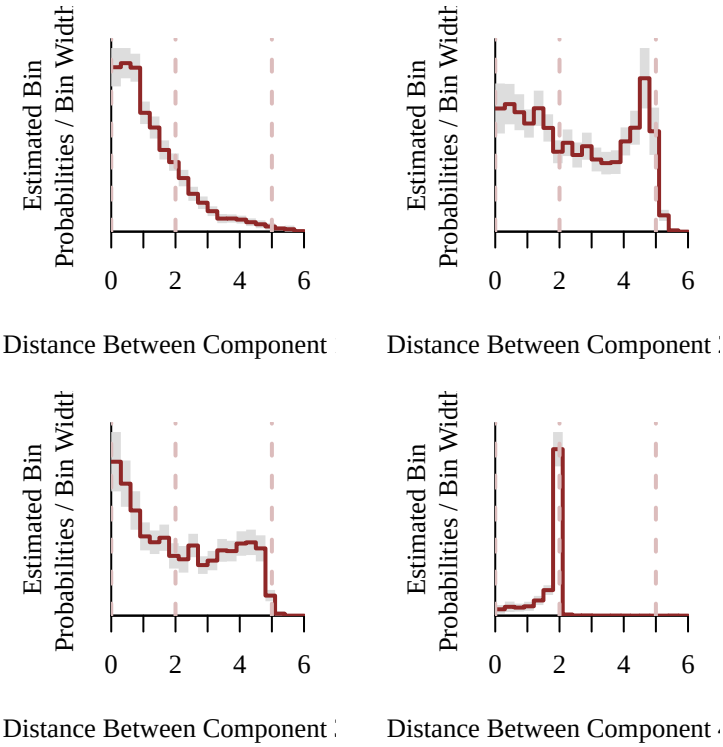
The true components in the simulation data generating process are centered at values of -4, 1, and 3 respectively. Consequently the distances between the true location parameters are 5 and 2. If two or more neighboring components in the mixture observational model collapse against each other at certain model configurations, however, then the distance between them will be zero, or as close to zero that the ordering constraint will allow.

In other words the pushforward posterior distribution for the separation between components should exhibit peaks at distances of 2 and 5. If we have extraneous components that collapse together at some model configurations then we should also see an additional peak at a distance of 0. Finally the location parameters of inactive components will not be coupled to the location parameters of any neighboring components, contributing a diffuse background beneath these peaks.

Indeed once we construct the pushforward posterior distributions for the distances we see exactly these behaviors.

```
var_repl <- list('mu'=array(sapply(1:5,
                            function(k) paste0("mu[", k, "]"))))
eval_component_separation <-
  list(function(mu) abs(mu[1] - mu[2]),
       function(mu) abs(mu[2] - mu[3]),
       function(mu) abs(mu[3] - mu[4]),
       function(mu) abs(mu[4] - mu[5]) )
names(eval_component_separation) <-
  sapply(1:4, function(k) paste0('d', k, k + 1))

component_separation <-
  util$eval_expectand_pushforwards(samples,
                                   eval_component_separation,
                                   var_repl)
```

```
par(mfrow=c(2, 2), mar=c(5, 5, 1, 1))

for (k in 1:4) {
  name <- paste0('d', k, k + 1)
  display_name <- paste('Distance Between Component', k, 'and', k + 1)
  util$plot_expectand_pushforward(component_separation[[name]],
                                  20, flim=c(0, 6),
                                  display_name=display_name)
  abline(v=0, lwd=2, lty=2, col=util$c_light)
  abline(v=2, lwd=2, lty=2, col=util$c_light)
  abline(v=5, lwd=2, lty=2, col=util$c_light)
}
```



Now that we understand the nature of these nasty degeneracies a bit better we might start to consider mediation strategies.

For example we might use a prior model for the component probabilities that concentrates on the simplex boundaries, encouraging the deactivation of any unnecessary components. That said we'd still have to contend with a degeneracy regarding which of the components are deactivated. Moreover we would have to carefully tune the prior model to ensure that we allow as many components to activate as needed to adequately model the observed data.

One approach to avoiding component collapse, indeed one that shows up over and over again in the statistics literature, is to employ *repulsive* prior models for the component locations. These prior models are designed to suppress model configurations where the component probability distributions are too close to each other. This approach is often used in tandem with other approaches, such as a sparsifying prior model for the component probabilities and/or ordering constraints.

Combining enough of these strategies together can absolutely yield reasonable results in some cases. To be honest, however, I have not found any combination of heuristics to be robust enough to make redundant mixture observational models a reliably productive tool in practice.

Beyond computational considerations redundant mixture observational models also suffer from fundamental interpretation issues. Because of the possibility of extraneous components we can-

not in general associate the individual component observational models with explicit features of the true data generating process. In particular we cannot interpret inferences for the component model configurations independently of the others. At that point we are not really learning about the structure of the true data generating process so much as learning particular patterns that happen to arise in particular observations.

For all of these reasons I avoid redundant mixture observational models as much as possible in my own analyses. I have found that building up mixture observational models from interpretable component observational models that can be tied to distinct aspects of the true data generating process tends to not only be more straightforward to implement but also yield more performant inferences and more generalizable predictions.

6 Conclusion

Mixture modeling is a general modeling technique that is useful in a diversity of practical applications. Here we have focused on relatively simple mixture models, but the basic structure provides a foundation for even more sophisticated models.

For example all of the examples presented in this chapter treat the component probabilities as parameters to be inferred. In some applications, however, it is more useful to *derive* them from the output of another part of the model. This approach can, for instance, allow the contribution of the component models to be mediated by external circumstances.

Similarly there's no reason why the component probabilities need to be static. In many applications it's natural for the component probabilities to vary across temporal or spatial dimensions. Modeling a sequence of evolving component probabilities results in some powerful modeling techniques, including the infamous hidden Markov model.

Finally individual component models can in theory be mixture models of their own. Indeed nested discrete mixture models can be interpreted as an implementation of conditional probability theory, allowing us to model data generating processes with conditional, but unobserved, logic.

Acknowledgements

I thank jd and EM Wolkovich for helpful comments.

A very special thanks to everyone supporting me on Patreon: Adam Fleischhacker, Adriano Yoshino, Alejandro Navarro-Martínez, Alessandro Varacca, Alex D, Alexander Noll, Alexander Rosteck, Andrea Serafino, Andrew Mascioli, Andrew Rouillard, Andrew Vigotsky, Ara Winter, Austin Rochford, Avraham Adler, Ben Matthews, Ben Swallow, Benoit Essiambre, Bertrand Wilden, Bradley Kolb, Brendan Galdo, Brynjolfur Gauti Jónsson, Cameron Smith, Canaan

Breiss, Cat Shark, CG, Charles Naylor, Chase Dwelle, Chris Jones, Christopher Mehrvarzi, Colin Carroll, Colin McAuliffe, Damien Mannion, dan mackinlay, Dan W Joyce, Dan Waxman, Dan Weitzenfeld, Daniel Edward Marthaler, Daniel Saunders, Darshan Pandit, Darthmaluus , David Galley, David Wurtz, Doug Rivers, Dr. Jobo, Dr. Omri Har Shemesh, Dylan Maher, Ed Cashin, Edgar Merkle, Eli Witus, Eric LaMotte, Ero Carrera, Eugene O'Friel, Felipe González, Fergus Chadwick, Finn Lindgren, Geoff Rollins, Håkan Johansson, Hamed Bastan-Hagh, haubur, Hector Munoz, Henri Wallen, hs, Hugo Botha, Ian, Ian Costley, idontgetoutmuch, Ignacio Vera, Ilaria Prosdocimi, Isaac Vock, Isidor Belic, J Michael Burgess, jacob pine, Jair Andrade, James C, James Hodgson, James Wade, Janek Berger, Jason Martin, Jason Pekos, jd, Jeff Burnett, Jeff Dotson, Jeff Helzner, Jeffrey Erlich, Jessica Graves, Joe Sloan, John Flournoy, Jonathan H. Morgan, Jonathon Vallejo, Joran Jongerling, JU, June, Justin Bois, Kádár András, Karim Naguib, Karim Osman, Kejia Shi, Kristian Gårdhus Wichmann, Lars Barquist, lizzie , Logan Sullivan, LOU ODETTE, Luís F, Marcel Lüthi, Marek Kwiatkowski, Mariana Carmona, Mark Donoghoe, Markus P., Márton Vaitkus, Matthew, Matthew Kay, Matthew Mulvahill, Matthieu LEROY, Mattia Arsendi, Maurits van der Meer, Michael Colaresi, Michael DeWitt, Michael Dillon, Michael Lerner, Mick Cooney, Mike Lawrence, N Sanders, N.S. , Name, Nathaniel Burbank, Nic Fishman, Nicholas Clark, Nicholas Cowie, Nick S, Octavio Medina, Ole Rogeberg, Oliver Crook, Olivier Ma, Patrick Kelley, Patrick Boehnke, Pau Pereira Batlle, Peter Johnson, Pieter van den Berg, ptr, Ramiro Barrantes Reynolds, Raúl Peralta Lozada, Ravin Kumar, Rémi , Rex Ha, Riccardo Fusaroli, Richard Nerland, Robert Frost, Robert Goldman, Robert kohn, Robin Taylor, Ryan Grossman, Ryan Kelly, S Hong, Sean Wilson, Sergiy Protsiv, Seth Axen, shira, Simon Duane, Simon Lilburn, Spencer Carter, sssz, Stan_user, Stephen Lienhard, Stew Watts, Stone Chen, Susan Holmes, Svilup, Tao Ye, Tate Tunstall, Tatsuo Okubo, Teresa Ortiz, Theodore Dasher, Thomas Siegert, Thomas Vladeck, Tobychev, Tomáš Frýda, Tony Wuersch, Virginia Fisher, Vladimir Markov, Wil Yegelwel, Will Farr, Will Lowe, Will^2, woejozney, Xianda Sun, yolhaj , yureq , Zach A, Zad Rafi, and Zhengchen Ca.

License

A repository containing all of the files used to generate this chapter is available on GitHub.

Original Computing Environment

```
writeLines(readLines(file.path(Sys.getenv("HOME"), ".R/Makevars")))
```

```
CC=clang

CXXFLAGS=-O3 -mtune=native -march=native -Wno-unused-variable -Wno-unused-function -Wno-macr
CXX=clang++ -arch x86_64 -ftemplate-depth-256

CXX14FLAGS=-O3 -mtune=native -march=native -Wno-unused-variable -Wno-unused-function -Wno-ma
CXX14=clang++ -arch x86_64 -ftemplate-depth-256
```

```
sessionInfo()
```

```
R version 4.3.2 (2023-10-31)
Platform: x86_64-apple-darwin20 (64-bit)
Running under: macOS Sonoma 14.4.1

Matrix products: default
BLAS:   /Library/Frameworks/R.framework/Versions/4.3-x86_64/Resources/lib/libRblas.0.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/4.3-x86_64/Resources/lib/libRlapack.dylib;

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

time zone: America/New_York
tzcode source: internal

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] colormap_0.1.4     rstan_2.32.6       StanHeaders_2.32.7

loaded via a namespace (and not attached):
 [1] gtable_0.3.4       jsonlite_1.8.8     compiler_4.3.2     Rcpp_1.0.11
 [5] stringr_1.5.1      parallel_4.3.2     gridExtra_2.3      scales_1.3.0
 [9] yaml_2.3.8         fastmap_1.1.1      ggplot2_3.4.4      R6_2.5.1
[13] curl_5.2.0         knitr_1.45         tibble_3.2.1       munsell_0.5.0
[17] pillar_1.9.0       rlang_1.1.2        utf8_1.2.4         V8_4.4.1
```

```
[21] stringi_1.8.3      inline_0.3.19      xfun_0.41          RcppParallel_5.1.7
[25] cli_3.6.2          magrittr_2.0.3     digest_0.6.33      grid_4.3.2
[29] lifecycle_1.0.4    vctrs_0.6.5        evaluate_0.23      glue_1.6.2
[33] QuickJSR_1.0.8     codetools_0.2-19   stats4_4.3.2       pkgbuild_1.4.3
[37] fansi_1.0.6        colorspace_2.1-0   rmarkdown_2.25     matrixStats_1.2.0
[41] tools_4.3.2        loo_2.6.0          pkgconfig_2.0.3    htmltools_0.5.7
```

Stan
Program 2 `signal\_background1.stan`

```
data {
  // Signal and background observations
  int<lower=1> N;
  array[N] real<lower=0> y;
}

parameters {
  real mu_signal;                // Signal location
  real<lower=0> sigma_signal;    // Signal scale
  real<lower=0> beta_back;       // Background scale
  real<lower=0, upper=1> lambda; // Background probability
}

model {
  // Prior model
  mu_signal ~ normal(50, 50 / 2.32);   // 0 <~ mu_signal    <~ 100
  sigma_signal ~ normal(0, 25 / 2.57); // 0 <~ sigma_signal <~  25
  beta_back ~ normal(0, 50 / 2.57);    // 0 <~ beta_back    <~  50
  // Implicit uniform prior density function for lambda

  // Observational model
  for (n in 1:N) {
    target += log_mix(lambda,
                      exponential_lpdf(y[n] | 1 / beta_back),
                      cauchy_lpdf(y[n] | mu_signal, sigma_signal));
  }
}

generated quantities {
  array[N] real<lower=0> y_pred = rep_array(-1, N);

  for (n in 1:N) {
    if (bernoulli_rng(lambda)) {
      y_pred[n] = exponential_rng(1 / beta_back);
    } else {
      while (y_pred[n] < 0) {
        y_pred[n] = cauchy_rng(mu_signal, sigma_signal);
      }
    }
  }
}
```

Stan
Program 3 `signal\_background2.stan`

```
data {
  // Signal and background observations
  int<lower=1> N;
  array[N] real<lower=0> y;
}

parameters {
  real mu_signal;                // Signal location
  real<lower=0> sigma_signal;    // Signal scale
  real<lower=0> beta_back;       // Background scale
  real<lower=0, upper=1> lambda; // Background probability
}

model {
  // Prior model
  mu_signal ~ normal(50, 50 / 2.32);   // 0 <~ mu_signal    <~ 100
  sigma_signal ~ normal(0, 25 / 2.57); // 0 <~ sigma_signal <~  25
  beta_back ~ normal(0, 50 / 2.57);    // 0 <~ beta_back    <~  50
  // Implicit uniform prior density function for lambda

  // Observational model
  for (n in 1:N) {
    target += log_mix(lambda,
                      exponential_lpdf(y[n] | 1 / beta_back),
                      cauchy_lpdf(y[n] | mu_signal, sigma_signal));
  }
}

generated quantities {
  array[N] real<lower=0, upper=1> p;
  array[N] int<lower=0, upper=1> z_pred;

  for (n in 1:N) {
    vector[2] xs = [   log(lambda)
                     + exponential_lpdf(y[n] | 1 / beta_back),
                       log(1 - lambda)
                     + cauchy_lpdf(y[n] | mu_signal, sigma_signal) ]';
    p[n] = softmax(xs)[1];
    z_pred[n] = bernoulli_rng(p[n]);
  }
}
```

Stan
Program 4 `simu\_zip.stan`

```
data {
  int<lower=1> N;   // Number of observations
  real<lower=0> mu; // Poisson intensity
  real<lower=0, upper=1> lambda; // Main component probability
}

generated quantities {
  // Initialize predictive variables with inflated value
  array[N] int<lower=0> y = rep_array(0, N);

  for (n in 1:N) {
    // If we sample the non-inflating component then replace initial
    // value with a Poisson sample
    if (bernoulli_rng(lambda)) {
      y[n] = poisson_rng(mu);
    }
  }
}
```

Stan
Program 5 `zip1.stan`

```
data {
  int<lower=1> N;          // Number of observations
  array[N] int<lower=0> y; // Positive integer observations
}

parameters {
  real<lower=0> mu;              // Poisson intensity
  real<lower=0, upper=1> lambda; // Main component probability
}

model {
  // Prior model
  mu ~ normal(0, 15 / 2.57); // 0 <~ mu <~ 15
  // Implicit uniform prior density function for lambda

  // Observational model
  for (n in 1:N) {
    if (y[n] == 0) {
      target += log_mix(lambda, poisson_lpmf(y[n] | mu), 0);
    } else {
      target += log(lambda) + poisson_lpmf(y[n] | mu);
    }
  }
}

generated quantities {
  // Initialize predictive variables with inflated value
  array[N] int<lower=0> y_pred = rep_array(0, N);

  for (n in 1:N) {
    // If we sample the non-inflating component then replace initial
    // value with a Poisson sample
    if (bernoulli_rng(lambda)) {
      y_pred[n] = poisson_rng(mu);
    }
  }
}
```

Stan
Program 6 `prior\_tune.stan`

```
functions {
  // Differences between inverse gamma tail
  // probabilities and target probabilities
  vector tail_delta(vector y, vector theta,
                    array[] real x_r, array[] int x_i) {
    vector[2] deltas;
    deltas[1] = inv_gamma_cdf(theta[1] | exp(y[1]), exp(y[2])) - 0.01;
    deltas[2] = 1 - inv_gamma_cdf(theta[2] | exp(y[1]), exp(y[2])) - 0.01;
    return deltas;
  }
}

data {
  real<lower=0>     y_low;
  real<lower=y_low> y_high;
}

transformed data {
  // Initial guess at inverse gamma parameters
  vector[2] y_guess = [log(2), log(5)]';
  // Target quantile
  vector[2] theta = [y_low, y_high]';
  vector[2] y;
  array[0] real x_r;
  array[0] int x_i;

  // Find inverse Gamma density parameters that ensure
  // 1% probability below y_low and 1% probability above y_high
  y = algebra_solver(tail_delta, y_guess, theta, x_r, x_i);

  print("alpha = ", exp(y[1]));
  print("beta = ", exp(y[2]));
}

generated quantities {
  real alpha = exp(y[1]);
  real beta = exp(y[2]);
}
```

Stan
Program 7 `zip2.stan`

```
data {
  int<lower=1> N;          // Number of observations
  array[N] int<lower=0> y; // Positive integer observations
}

parameters {
  real<lower=0> mu;              // Poisson intensity
  real<lower=0, upper=1> lambda; // Main component probability
}

model {
  // Prior model
  mu ~ inv_gamma(3.5, 9.0); // 1   <~   mu   <~ 15
  // Implicit uniform prior density function for lambda

  // Observational model
  for (n in 1:N) {
    if (y[n] == 0) {
      target += log_mix(lambda, poisson_lpmf(y[n] | mu), 0);
    } else {
      target += log(lambda) + poisson_lpmf(y[n] | mu);
    }
  }
}

generated quantities {
  // Initialize predictive variables with inflated value
  array[N] int<lower=0> y_pred = rep_array(0, N);

  for (n in 1:N) {
    // If we sample the non-inflating component then replace initial
    // value with a Poisson sample
    if (bernoulli_rng(lambda)) {
      y_pred[n] = poisson_rng(mu);
    }
  }
}
```

Stan
Program 8 `simu\_zoib.stan`

```
data {
  int<lower=1> N; // Number of observations
}

transformed data {
  real alpha = 3;
  real beta = 2;
  simplex[3] lambda = [0.75, 0.15, 0.10]';
}

generated quantities {
  // Initialize predictive variables with inflated value
  array[N] real<lower=0, upper=1> y;

  for (n in 1:N) {
    int z = categorical_rng(lambda);

    if (z == 1) {
      y[n] = beta_rng(alpha, beta);
    } else if (z == 2) {
      y[n] = 0;
    } else {
      y[n] = 1;
    }
  }
}
```

Stan
Program 9 `zoib1.stan`

```
data {
  int<lower=1> N;                     // Number of observations
  array[N] real<lower=0, upper=1> y; // Unit-interval valued observations
}

transformed data {
  int<lower=0> N_zero = 0;
  int<lower=0> N_one = 0;
  int<lower=0> N_else = N;

  for (n in 1:N) {
    if (y[n] == 0) N_zero += 1;
    if (y[n] == 1) N_one  += 1;
  }

  N_else -= N_one + N_zero;
}

parameters {
  real<lower=0> alpha; // Beta shape
  real<lower=0>  beta; // Beta scale
  simplex[3]   lambda; // Component probabilities
}

model {
  // Prior model
  alpha ~ normal(0, 10 / 2.57); // 0 <~ alpha <~ 10
  beta ~ normal(0, 10 / 2.57);  // 0 <~ beta  <~ 10
  // Implicit uniform prior density function for lambda

  // Observational model
  target += multinomial_lpmf({N_else, N_zero, N_one} | lambda);

  for (n in 1:N) {
    if (0 < y[n] && y[n] < 1) {
      target += beta_lpdf(y[n] | alpha, beta);
    }
  }
}

generated quantities {
  array[N] real<lower=0, upper=1> y_pred;

  for (n in 1:N) {
    int z = categorical_rng(lambda);

    if (z == 1) {
      y_pred[n] = beta_rng(alpha, beta);
    } else if (z == 2) {
      y_pred[n] = 0;
```

Stan
Program 10 `zoib2a.stan`

```
data {
  // Number of non-zero/one observations
  int<lower=1> N_else;
  // Non-zero/one observations
  array[N_else] real<lower=0, upper=1> y_else;
}

parameters {
  real<lower=0> alpha; // Beta shape
  real<lower=0>  beta; // Beta scale
}

model {
  // Prior model
  alpha ~ normal(0, 10 / 2.57); // 0 <~ alpha <~ 10
  beta ~ normal(0, 10 / 2.57);  // 0 <~ beta  <~ 10

  // Observational model
  target += beta_lpdf(y_else | alpha, beta);
}
```

Stan
Program 11 `zoib2b.stan`

```
data {
  int<lower=1> N_zero; // Number of zero observations
  int<lower=1> N_one;  // Number of one observations
  int<lower=1> N_else; // Number of non-zero/one observations
}

parameters {
  simplex[3] lambda; // Component probabilities
}

model {
  // Prior model
  // Implicit uniform prior density function for lambda

  // Observational model
  target += multinomial_lpmf({N_else, N_zero, N_one} | lambda);
}
```

Stan
Program 12 simu_normal_mix.stan

```
data {
  int<lower=1> N;      // Number of observations
}

transformed data {
  int K = 3;                                    // Number of components
  array[K] real mu = {-4, 1, 3};                // Component locations
  array[K] real<lower=0> sigma = {2, 0.5, 0.5}; // Component scales
  simplex[K] lambda = [0.3, 0.5, 0.2]';         // Component probabilities

}

generated quantities {
  array[N] real y;

  for (n in 1:N) {
    int z = categorical_rng(lambda);
    y[n] = normal_rng(mu[z], sigma[z]);
  }
}
```

Stan
Program 13 `normal\_mix1.stan`

```
data {
  int<lower=1> N;  // Number of observations
  array[N] real y; // Observations
}

transformed data {
  int K = 3;                                      // Number of components
  array[K] real mu = {-4, 1, 3};                  // Component locations
  array[K] real<lower=0> sigma = {2, 0.5, 0.5}; // Component scales
}

parameters {
  simplex[K] lambda; // Component probabilities
}

model {
  // Prior model
  // Implicit uniform prior density function for lambda

  // Observational model
  for (n in 1:N) {
    vector[K] lpds;
    for (k in 1:K) {
      lpds[k] = log(lambda[k]) + normal_lpdf(y[n] | mu[k], sigma[k]);
    }
    target += log_sum_exp(lpds);
  }
}

generated quantities {
  array[N] real y_pred;

  for (n in 1:N) {
    int z = categorical_rng(lambda);
    y_pred[n] = normal_rng(mu[z], sigma[z]);
  }
}
```

Stan
Program 14 normal_mix2a.stan

```
data {
  int<lower=1> N;  // Number of observations
  array[N] real y; // Observations
}

transformed data {
  int K = 3;                                        // Number of components
  array[K] real<lower=0> sigma = {2, 0.5, 0.5}; // Component scales
}

parameters {
  array[K] real mu;  // Component locations
  simplex[K] lambda; // Component probabilities
}

model {
  // Prior model
  mu ~ normal(0, 10 / 2.32); // -10 <~ mu[k] <~ +10
  // Implicit uniform prior density function for lambda

  // Observational model
  for (n in 1:N) {
    vector[K] lpds;
    for (k in 1:K) {
      lpds[k] = log(lambda[k]) + normal_lpdf(y[n] | mu[k], sigma[k]);
    }
    target += log_sum_exp(lpds);
  }
}

generated quantities {
  array[N] real y_pred;

  for (n in 1:N) {
    int z = categorical_rng(lambda);
    y_pred[n] = normal_rng(mu[z], sigma[z]);
  }
}
```

Stan
Program 15 normal_mix2b.stan

```
data {
  int<lower=1> N;  // Number of observations
  array[N] real y; // Observations
}

transformed data {
  int K = 3;                                      // Number of components
  array[K] real<lower=0> sigma = {2, 0.5, 0.5}; // Component scales
}

parameters {
  array[K] real mu;  // Component locations
  simplex[K] lambda; // Component probabilities
}

model {
  // Prior model
  mu[1] ~ normal(-4, 2 / 2.32); // -6 <~ mu[2] <~ -2
  mu[2] ~ normal( 0, 2 / 2.32); // -2 <~ mu[2] <~ +2
  mu[3] ~ normal(+4, 2 / 2.32); // +2 <~ mu[2] <~ +6
  // Implicit uniform prior density function for lambda

  // Observational model
  for (n in 1:N) {
    vector[K] lpds;
    for (k in 1:K) {
      lpds[k] = log(lambda[k]) + normal_lpdf(y[n] | mu[k], sigma[k]);
    }
    target += log_sum_exp(lpds);
  }
}

generated quantities {
  array[N] real y_pred;

  for (n in 1:N) {
    int z = categorical_rng(lambda);
    y_pred[n] = normal_rng(mu[z], sigma[z]);
  }
}
```

Stan
Program 16 `normal\_mix2c.stan`

```
data {
  int<lower=1> N;  // Number of observations
  array[N] real y; // Observations
}

transformed data {
  int K = 3;                                        // Number of components
  array[K] real<lower=0> sigma = {2, 0.5, 0.5}; // Component scales
}

parameters {
  ordered[K] mu;     // Component locations
  simplex[K] lambda; // Component probabilities
}

model {
  // Prior model
  mu ~ normal(0, 10 / 2.32); // -10 <~ mu[k] <~ +10
  // Implicit uniform prior density function for lambda

  // Observational model
  for (n in 1:N) {
    vector[K] lpds;
    for (k in 1:K) {
      lpds[k] = log(lambda[k]) + normal_lpdf(y[n] | mu[k], sigma[k]);
    }
    target += log_sum_exp(lpds);
  }
}

generated quantities {
  array[N] real y_pred;

  for (n in 1:N) {
    int z = categorical_rng(lambda);
    y_pred[n] = normal_rng(mu[z], sigma[z]);
  }
}
```

Stan
Program 17 `normal\_mix3a.stan`

```
data {
  int<lower=1> N;  // Number of observations
  array[N] real y; // Observations
}

transformed data {
  int K = 3; // Number of components
}

parameters {
  array[K] real mu;              // Component locations
  array[K] real<lower=0> sigma; // Component scales
  simplex[K] lambda;             // Component probabilities
}

model {
  // Prior model
  mu ~ normal(0, 10 / 2.32);    // -10 <~  mu[k]   <~ +10
  sigma ~ normal(0, 10 / 2.57); //   0 <~ sigma[k] <~ +10

  // Implicit uniform prior density function for lambda

  // Observational model
  for (n in 1:N) {
    vector[K] lpds;
    for (k in 1:K) {
      lpds[k] = log(lambda[k]) + normal_lpdf(y[n] | mu[k], sigma[k]);
    }
    target += log_sum_exp(lpds);
  }
}

generated quantities {
  array[N] real y_pred;

  for (n in 1:N) {
    int z = categorical_rng(lambda);
    y_pred[n] = normal_rng(mu[z], sigma[z]);
  }
}
```

Stan
Program 18 normal_mix3b.stan

```
data {
  int<lower=1> N;  // Number of observations
  array[N] real y; // Observations
}

transformed data {
  int K = 3; // Number of components
}

parameters {
  ordered[K] mu;                // Component locations
  array[K] real<lower=0> sigma; // Component scales
  simplex[K] lambda;            // Component probabilities
}

model {
  // Prior model
  mu ~ normal(0, 10 / 2.32);    // -10 <~  mu[k]   <~ +10
  sigma ~ normal(0, 10 / 2.57); //   0 <~ sigma[k] <~ +10

  // Implicit uniform prior density function for lambda

  // Observational model
  for (n in 1:N) {
    vector[K] lpds;
    for (k in 1:K) {
      lpds[k] = log(lambda[k]) + normal_lpdf(y[n] | mu[k], sigma[k]);
    }
    target += log_sum_exp(lpds);
  }
}

generated quantities {
  array[N] real y_pred;

  for (n in 1:N) {
    int z = categorical_rng(lambda);
    y_pred[n] = normal_rng(mu[z], sigma[z]);
  }
}
```

Stan
Program 19 normal_mix4.stan

```
data {
  int<lower=1> N;  // Number of observations
  array[N] real y; // Observations

  int K; // Number of components
}

parameters {
  ordered[K] mu;                // Component locations
  array[K] real<lower=0> sigma; // Component scales
  simplex[K] lambda;            // Component probabilities
}

model {
  // Prior model
  mu ~ normal(0, 10 / 2.32);    // -10 <~  mu[k]   <~ +10
  sigma ~ normal(0, 10 / 2.57); //   0 <~ sigma[k] <~ +10

  // Implicit uniform prior density function for lambda

  // Observational model
  for (n in 1:N) {
    vector[K] lpds;
    for (k in 1:K) {
      lpds[k] = log(lambda[k]) + normal_lpdf(y[n] | mu[k], sigma[k]);
    }
    target += log_sum_exp(lpds);
  }
}

generated quantities {
  array[N] real y_pred;

  for (n in 1:N) {
    int z = categorical_rng(lambda);
    y_pred[n] = normal_rng(mu[z], sigma[z]);
  }
}
```